VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Průmysl 4.0 - Optimalizace sběru online dat pro průmyslové roboty

Industry 4.0 - Optimization of Online Data Collection for Industrial Robots

2020 Jakub Kačík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Kačík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Průmysl 4.0 - Optimalizace sběru online dat pro průmyslové roboty
Industry 4.0 - Optimization of Online Data Collection for Industrial Robots

Jazyk vypracování: slovenština

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat softwarové řešení sběru dat z průmyslových robotů nazávisle na typu monitorované platformy a jeho testování pro použití v produkčním prostředí. Řešení zabezpečí optimalizaci výkonu platformy, uchová data a zpřístupní je prostřednictvím API rozhraní.

1. Student provede analýzu současného stavu.
2. Student navrhne řešení sběru dat nezávisle na typu platformy a vybere vhodné zařízení pro server.
3. Student implementuje webový server pro zobrazení získávaných dat a API rozhraní pro připojení mobilních zařízení a upraví existující iOS aplikaci pro nový typ komunikace.
4. Student navrhne a provede testování výsledného řešení.
5. Student zhodnotí dosažené výsledky.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Patrik Urban**

Konzultant diplomové práce: Ing. Jan Kožusznik, Ph.D.

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA fakulta elektrotechniky a informatiky 1D14 OSTRAVA

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 30. apríla 2020

Súhlasím so zverejnením tejto diplomovej práce podľa požiadaviek čl . 26, odst. 9 Študijného a skúšobného poriadku pre štúdium v magisterských programoch VŠB-TU Ostrava.

V Ostrave 30. apríla 2020 .....

Rád by som predovšetkým poďakoval vedúcemu diplomovej práce Ing. Patrikovi Urbanovi, za príkladné vedenie, cenné rady a všetok venovaný čas. Ďakujem taktiež pánovi Ing. Janu Kožusznikovi, Ph.D. za ochotu a pomoc pri riešení otázok ohľadom návrhu informačného systému. V neposlednom rade ďakujem svojej rodine a priateľom za podporu počas celého štúdia.

Abstrakt

Táto diplomová práca sa zaoberá návrhom a implementáciou webového servera pre zber dát z priemyselných robotov od spoločnosti ABB. Cieľom práce je vytvorenie nástroja, ktorý by zabezpečil nepretržitý zber dát, ich uloženie, vizualizáciu a poskytnutie jednotlivým klientom prostredníctvom API rozhrania. Teoretická časť popisuje rozhranie pre získavanie dát z robotov, existujúce aplikácie a ich hlavné nedostatky. Praktická časť zoznamuje čitateľa s jednotlivými fázami návrhu systému, následnou implementáciou, testovaním a výsledným nasadením. Výstupom práce je plne funkčný nástroj pre monitorovanie robotov. Súčasťou je aj rozsiahle výkonnostné testovanie, aby sa overila použiteľnosť výsledného riešenia v produkčnom prostredí pri predpokladanej záťaži.

Kľúčové slová: informačný systém, webový server, robot, zber dát, Priemysel 4.0, .NET Core, React, JMeter, SonarQube, Docker, Azure IoT Edge, Azure DevOps, PostgreSQL

Abstract

This diploma thesis deals with the design and implementation of a web server for data collection from ABB company industrial robots. The aim of this thesis is to create a tool which would ensure continuous data collection, storage, visualization and provision this data to individual clients through the API interface. The theoretical section describes an interface for obtaining data from robots, existing applications and their main shortcomings. The practical section acquaints the reader with various stages of system design, implementation, testing and the final deployment. The outcome of the thesis is a fully functional tool for monitoring robots. Extensive performance testing is also included to verify the applicability of the final solution in a production environment at the anticipated load.

Keywords: information system, web server, robot, data collection, Industry 4.0, .NET Core, React, JMeter, SonarQube, Docker, Azure IoT Edge, Azure DevOps, PostgreSQL

Obsah

Zoznam použitých skratiek a symbolov

AOP	–	Aspect Oriented Programming
API	–	Application Programming Interface
ARM	–	Advanced RISC Machine
CD	–	Continuous Delivery
CI	–	Continuous Integration
CPU	–	Central Processing Unit
CRUD	–	Create, Read, Update, Delete
CSS	–	Cascading Style Sheets
DDD4	–	Double Data Rate 4
ER	–	Entity Relationship
FTP	–	File Transfer Protocol
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
IO	–	Input/Output
IoC	–	Inversion of Control
IP	–	Internet Protocol
JSON	–	JavaScript Object Notation
JWT	–	JSON Web Token
LPDDR2	–	Low-Power Double Data Rate 2
MD5	–	Message-Digest algorithm 5
mDNS	–	multicast Domain Name System
ORM	–	Object-Relational Mapping
OS	–	Operating System
RAM	–	Random Access Memory
REST	–	Representational State Transfer
RISC	–	Reduced Instruction Set Computer
SD	–	Secure Digital
SMS	–	Short Message Service
SQL	–	Structured Query Language
SSD	–	Solid State Drive
TCP	–	Transmission Control Protocol
URL	–	Uniform Resource Locator
XHTML	–	Extensible Hypertext Markup Language
XML	–	Extensible Markup Language

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov zdrojového kódu

Úvod

V dnešnej dobe si už takmer nevieme predstaviť život bez pripojenia na internet. Jednotlivci, malé spoločnosti ale aj veľké organizácie prichádzajú do kontaktu s informačnými systémami na dennom poriadku a častokrát bez toho, aby si to priamo uvedomovali. Tieto systémy sa stali neoddeliteľnou súčasťou každého z nás a našli si uplatnenie vo všetkých odvetviach priemyslu. Práve priemysel prechádza v posledných rokoch veľkými zmenami a čoraz častejšie sa skloňuje slovo Priemysel 4.0. S týmto výrazom sa spájajú pojmy ako automatizácia, analýza dát a neustála optimalizácia výrobného procesu. Čoraz väčší počet firiem chce mať pod kontrolou všetky aspekty svojej výroby a preto investujú nemalé finančné prostriedky do informačných systémov, ktoré im pri správnom aplikovaní môžu dopomôcť k lepším ekonomickým výsledkom.

Spoločnosť ABB s.r.o. predstavuje švédsko-švajčiarsku nadnárodnú spoločnosť so sídlom v Zürichu, ktorá zamestnáva cez 135 000 ľudí v približne 100 krajinách sveta. Zameriava sa prevažne na oblasť energetiky a automatizácie, a snaží sa neustále prinášať nové technológie. S týmto súvisí aj snaha implementovať do svojich produktov systémy, ktoré by uľahčili ich používanie a prepojili ich do jedného inteligentného celku.

Prvá kapitola sa bude najskôr zaoberať popisom rozhrania, pomocou ktorého je možné získavať širokú škálu dát z priemyselných robotov od spoločnosti ABB. V tejto časti budú detailnejšie popísané možnosti komunikácie s týmto rozhraním a taktiež rozdiely medzi jednotlivými verziami. Ďalej budú predstavené existujúce aplikácie využívajúce toto rozhranie a ich hlavné nedostatky, ktoré stoja za vytvorením tejto diplomovej práce.

Ďalšia kapitola sa bude venovať návrhu informačného systému pre monitorovanie robotov. Najskôr dôjde k analýze podobných aplikácií a následne bude vytvorený návrh na základe viacerých pohľadov na systém. Ten bude prechádzať postupne od jednotlivých prípadov užitia, cez rozdelenie softvéru do logických vrstiev až po detailný popis kľúčových častí systému. Taktiež sa tu bude nachádzať zoznam technológií použitých pri vývoji a výber servera pre nasadenie.

Nasledujúca kapitola sa bude venovať samotnej implementácií webového servera a taktiež úprave existujúcich aplikácií pre nový spôsob komunikácie. Budú tu popísané jednotlivé techniky programovania, ako aj použité knižnice. Text bude doplnený o ukážky zdrojových kódov pre najpodstatnejšie časti systému.

V predposlednej kapitole budú popísané jednotlivé spôsoby testovania aplikované na systém. Prvá časť sa bude zaoberať unit testovaním a použitými knižnicami. Druhá časť bude venovaná výkonnostnému testovaniu. Dôjde tu k popisu testovacieho prostredia, použitých nástrojov a v neposlednom rade samotnej metodiky testovania. Výsledky meraní budú doplnené o grafy, ktoré dajú čitateľovi lepšiu predstavu o vyťažení zdrojov servera a dobe odozvy pri narastajúcej záťaži. Posledná časť sa bude zaoberať statickou analýzou kódu.

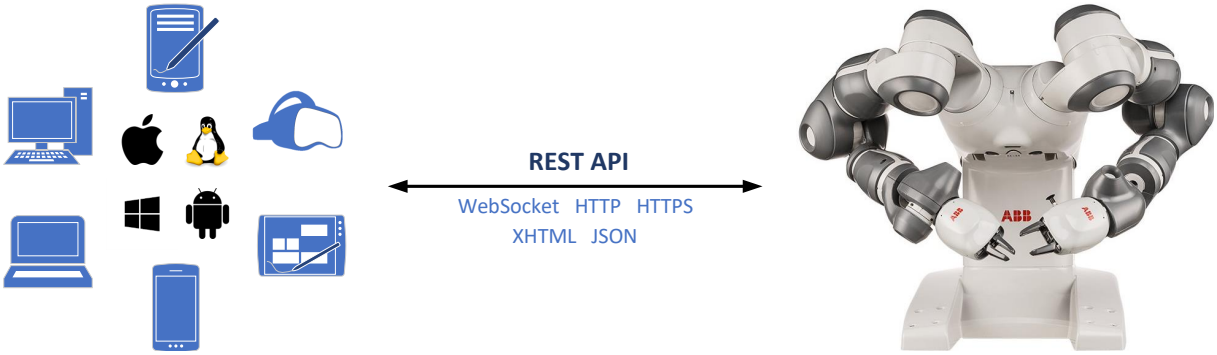
Záverečná kapitola bude venovaná nasadeniu výsledného riešenia na fyzický server. Bude sa tu nachádzať popis nástrojov, pomocou ktorých bude zabezpečená správa životného cyklu aplikácie, od vykonania zmien, cez zostavenie až po samotné nasadenie.

1 Súčasný stav

V tejto kapitole sa najskôr zaoberám popisom REST API rozhrania, pomocou ktorého je možné získavať širokú škálu dát z priemyselných robotov od spoločnosti ABB. Následne popisujem aktuálne softvérové riešenie a problémy spojené s danou implementáciou.

1.1 Robot Web Services

Robot Web Services predstavuje platformu, ktorá umožňuje vývojárom komunikovať s radičom robota prostredníctvom REST API rozhrania. Princíp komunikácie je zobrazený na Obrázku 1. Prvýkrát sa táto platforma objavila ako súčasť RobotWare[1] 6.0 a je označovaná ako verzia 1.0. S príchodom RobotWare 7.0 došlo k vydaniu verzie 2.0. Robot Web Services podporuje komunikáciu prostredníctvom *HTTP* a *WebSocket*[2] protokolu. Dáta sú získavané vo formáte *XHTML* alebo *JSON*. Každé volanie API musí byť autorizované. [1] [2]



Obr. 1: Robot Web Services komunikácia

1.1.1 Robot Web Services 1.0 vs 2.0

Tabuľka 1 obsahuje zhrnutie a porovnanie základných vlastností jednotlivých verzií.

Tabuľka 1: Porovnanie vlastností Robot Web Services 1.0 a 2.0

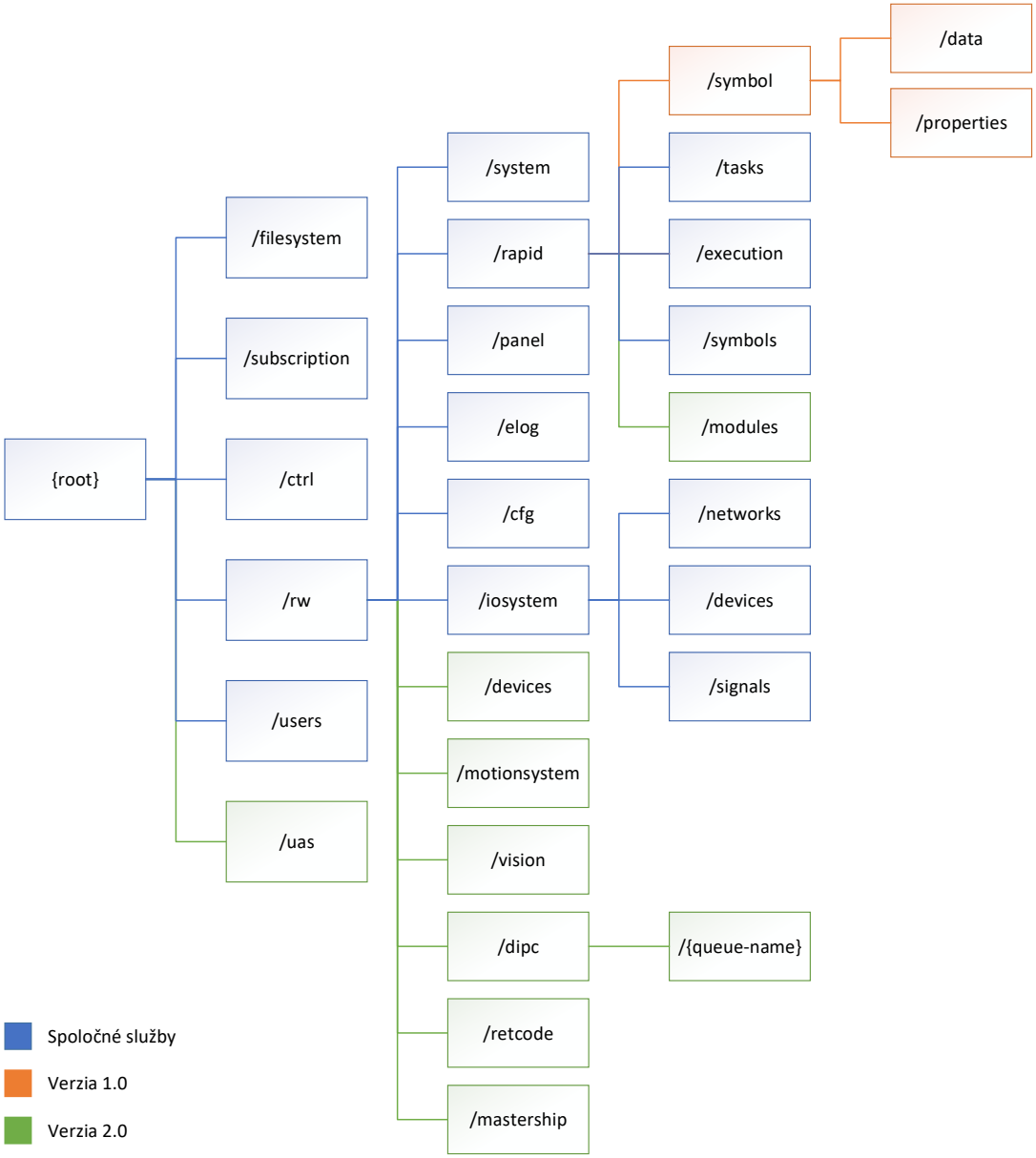
	1.0	2.0
Komunikačný protokol	HTTP/WebSocket	HTTPS/WebSocket
Formát odpovede	XHTML/JSON	XHTML/JSON
Typ zabezpečenia	Digest	Basic
API štandard	×	Open API
RobotWare	6.0 - 6.10	7.0

[1] RobotWare predstavuje riadiaci softvér robota.

[2] Trvalé obojstranné TCP pripojenie medzi klientom a serverom.

1.1.2 Služby

Robot Web Services sa skladá zo služieb, ktoré sú usporiadané do stromovej štruktúry. Táto štruktúra sa líši v závislosti od verzie. Porovnanie oboch verzií je zobrazené na Obrázku 2.



Obr. 2: Porovnanie poskytovaných služieb Robot Web Services

Hlavné skupiny dostupných služieb sú:

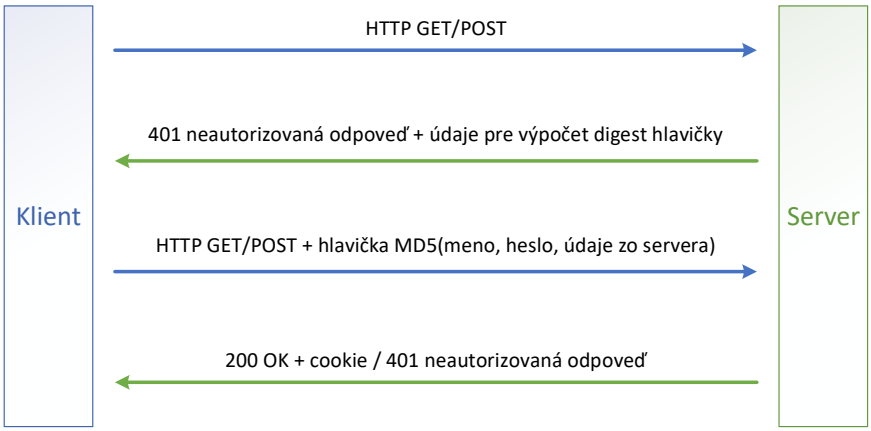
- **Filesystem:** Poskytuje vzdialený prístup k súborom a adresárom.
- **Subscription:** Všetky operácie spojené s WebSocket protokolom.
- **Ctrl:** Informácie o radiči robota.

- **Users:** Registrácia klientov.
- **RW:** Informácie o alarmoch, vstupno-výstupných operáciách, signáloch a podobne.
- **UAS:** Operácie spojené s autentifikáciou užívateľa.

1.1.3 Autentifikácia

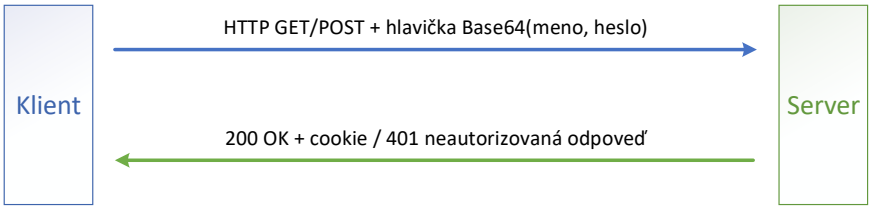
Každé zariadenie využívajúce služby Robot Web Services sa musí autentifikovať pomocou *mena* a *hesla*. V závislosti od verzie sa využíva *Digest* alebo *Basic* autentifikácia. Po úspešnom autentifikovaní obdrží klient od robota *cookie*, ktoré sú ďalej využívané pre každú požiadavku a udržanie relácie.

Verzia 1.0
Robot Web Services vo verzií 1.0 využíva Digest autentifikáciu, ktorá pozostáva z krokov zobrazených na Obrázku 3.



Obr. 3: Digest autentifikácia

Verzia 2.0
Pri verzií 2.0 došlo k zmene autentifikácie z Digest na Basic, ktorej princíp je popísaný na Obrázku 4.



Obr. 4: Basic autentifikácia

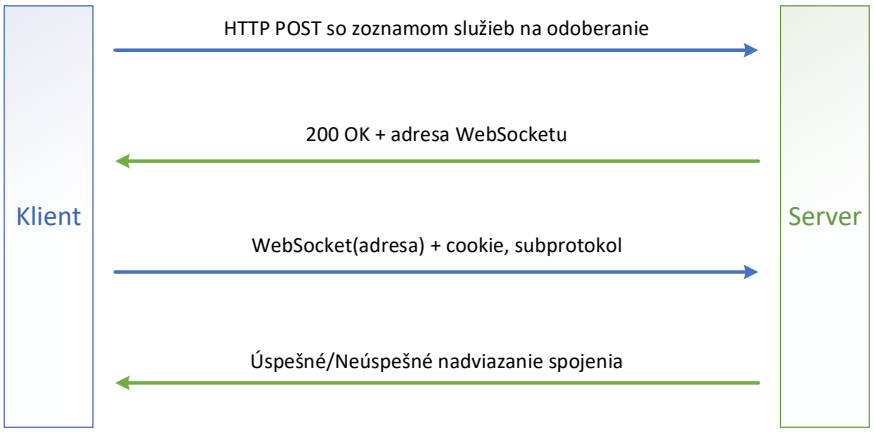
Táto metóda nešifruje užívateľské meno a heslo ale počíta s tým, že prenos medzi serverom a klientom je zabezpečený.

1.1.4 Relácia

Relácia medzi klientom a serverom je udržiavaná pomocou cookie. Server je schopný nadviazať až sedemdesiat relácií. Po dosiahnutí tohoto počtu je požiadavka o ďalšie pripojenie zamietnutá. Z jednej IP adresy je možné nadviazať maximálne päť relácií, pričom každá relácia môže obsahovať jedno WebSocket a dve HTTP spojenia. Po prerušení relácie dôjde k odhláseniu klienta a uvoľneniu všetkých zdrojov.

1.1.5 WebSocket

WebSocket protokol umožňuje klientovi prihlásiť sa na odber rôznych zdrojov robota. Táto funkcionalita je poskytovaná prostredníctvom služby *subscription*. Výhodou je, že robot sám informuje klienta odoslaním dát a nie je potrebné neustále dotazovanie ako v prípade HTTP požiadaviek. Odpoveď je vo formáte XHTML. Klient môže vytvoriť maximálne jedno zabezpečené WebSocket pripojenie, ktoré môže byť rozdelené do dvoch skupín. Táto služba umožňuje vytvoriť celkovo tisíc odberov pre všetkých pripojených klientov. Pre nadviazanie WebSocket pripojenia sú potrebné kroky z Obrázka 5.



Obr. 5: Nadviazanie WebSocket komunikácie

Robot Web Services umožňuje pre každý odber nastaviť prioritu, na základe ktorej bude server odosielať udalosti ku klientovi. Typy priorít sú:

- **Vysoká:** Udalosti odosielané ihneď po výskyte. Dostupná pre RAPID[3] a IO signály.
- **Stredná:** Maximálne oneskorenie 0.2 sekundy.
- **Nízka:** Maximálne oneskorenie 5 sekúnd.

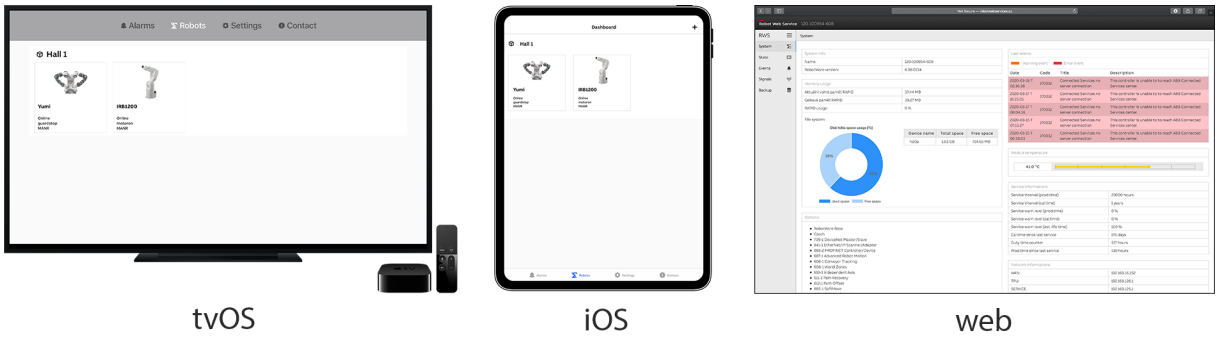
[3]RAPID predstavuje vysokoúrovňový programovací jazyk pre riadenie ABB robotov.

1.1.6 Vyhľadávanie robotov

Na vyhľadávanie robotov v lokálnej sieti je možné využiť *mDNS* protokol. Roboty ABB sú jedinečne identifikovateľné pod názvom *_http._tcp,rws*. Princíp spočíva v odoslaní multicast[4] správy, ktorá požiada robotov s daným názvom aby sa identifikovali. Tie následne vrátia IP adresu, na ktorej sa nachádzajú. [3]

1.2 Robot Web Service aplikácie

Na Obrázku 6 môžete vidieť platformy podporované aplikáciami Robot Web Service.



Obr. 6: Robot Web Service platformy

1.2.1 Webová stránka

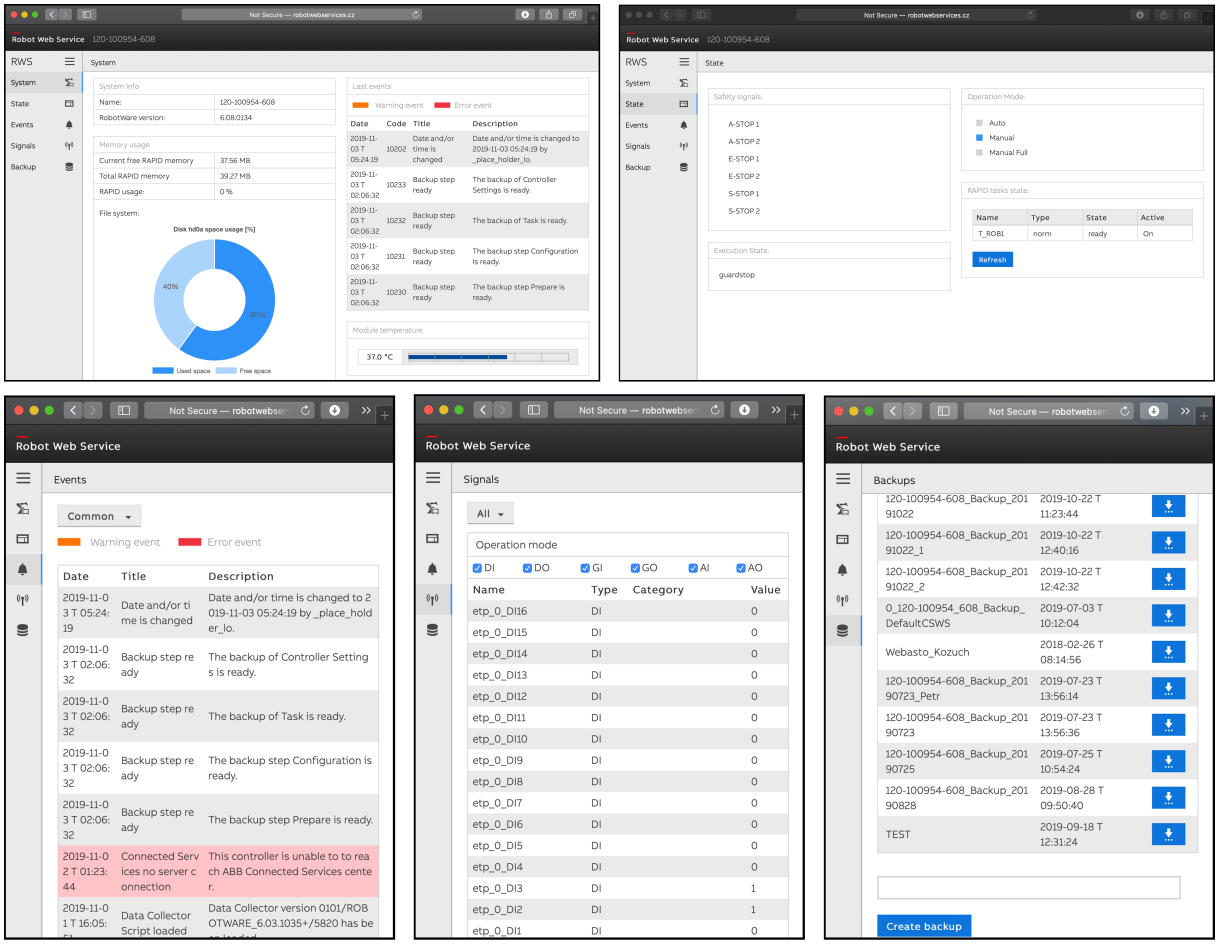
Toto riešenie kombinuje programovacie jazyky HTML, JavaScript a CSS. Stránka je hosťovaná priamo na robotovi, vďaka vstavanému webovému serveru[5]. Pri prvom načítaní je užívateľ vyzvaný k zadaniu užívateľského mena a hesla. Po úspešnom prihlásení začne aplikácia získavať dáta z robota pomocou HTTP požiadaviek. Webová stránka neukladá žiadne dáta a nevyužíva WebSocket komunikáciu. Aplikácia je rozdelená do nasledujúcich celkov:

- **Systém:** Poskytuje informácie o verzii RobotWare, type robota, aktuálnej teplote, využití pamäte, inštalovaných prvkoch, servisných intervaloch a pripojenej sieti.
- **Stav:** Zobrazuje prevádzkový režim robota, bezpečnostné okruhy a aktuálne bežiacu úlohu.
- **Alarmy:** Prehľad aktuálnych alarmov robota spolu s filtrovaním na základe kategórií.
- **Signály:** Zobrazuje signály robota a umožňuje ich filtráciu na základe módu a kategórie.
- **Záloha:** Slúži na vytváranie záloh systému robota a poskytuje ich prehľad.

Ukážka webovej stránky je zobrazená na Obrázku 7.

[4]Multicast umožňuje odoslanie paketov na všetky zariadenia v sieti.

[5]Webový server je dostupný pre robotov od verzie RobotWare 6.0.



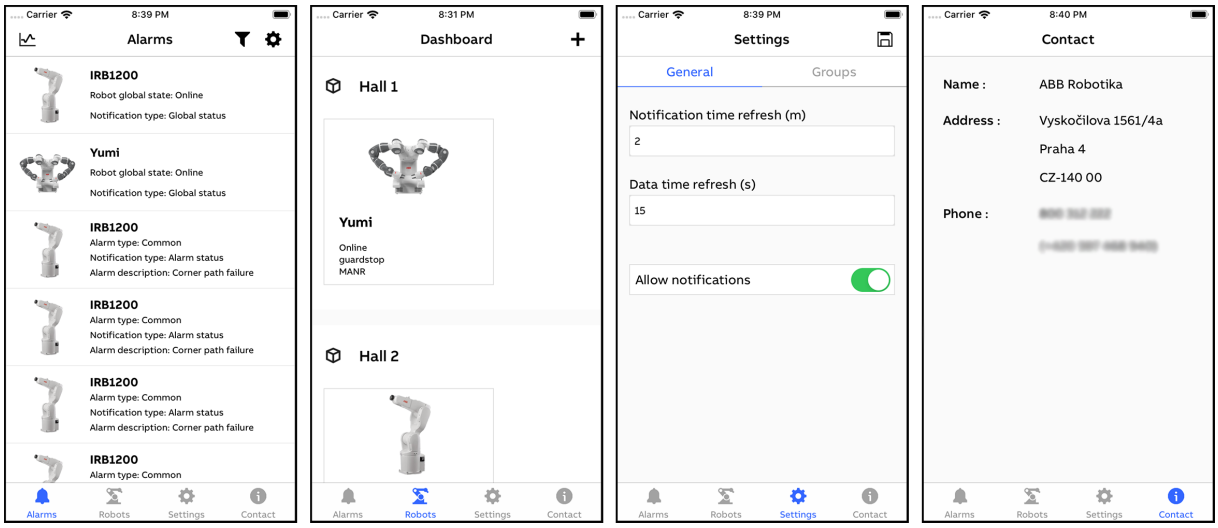
Obr. 7: Ukážka webovej aplikácie Robot Web Service

1.2.2 iOS

Jedná sa o aplikáciu pre iOS zariadenia, napísanú v jazyku Swift. Aplikácia umožňuje správu robotov a ich rozdelenie do skupín. Ďalej poskytuje informácie o stave pripojenia a nových alarmoch. Podrobnejšie informácie o robotoch sú zobrazované prostredníctvom webovej stránky, popísanej v Sekcii 1.2.1. Pre plnú funkcionalitu je potrebné, aby na robotovi táto stránka bežala. V aplikácií dochádza k lokálnemu ukladaniu alarmov pre jednotlivých robotov. Dáta sú získavané prostredníctvom HTTP požiadaviek. Aplikácia je rozdelená do nasledujúcich častí:

- **Alarmy:** Prehľad všetkých alarmov s možnosťou filtrovania a zobrazenia podrobností.
- **Roboty:** Slúži na zobrazenie skupín a robotov. Umožňuje pridať nového robota alebo zobraziť si jeho detaily spolu s možnosťou úpravy a vymazania.
- **Nastavenia:** Skladajú sa z dvoch častí. Prvá poskytuje nastavenia pre notifikácie a sťahovanie dát, druhá slúži na správu skupín.
- **Kontakt:** Zobrazenie kontaktných informácií o vlastníkovi produktu.

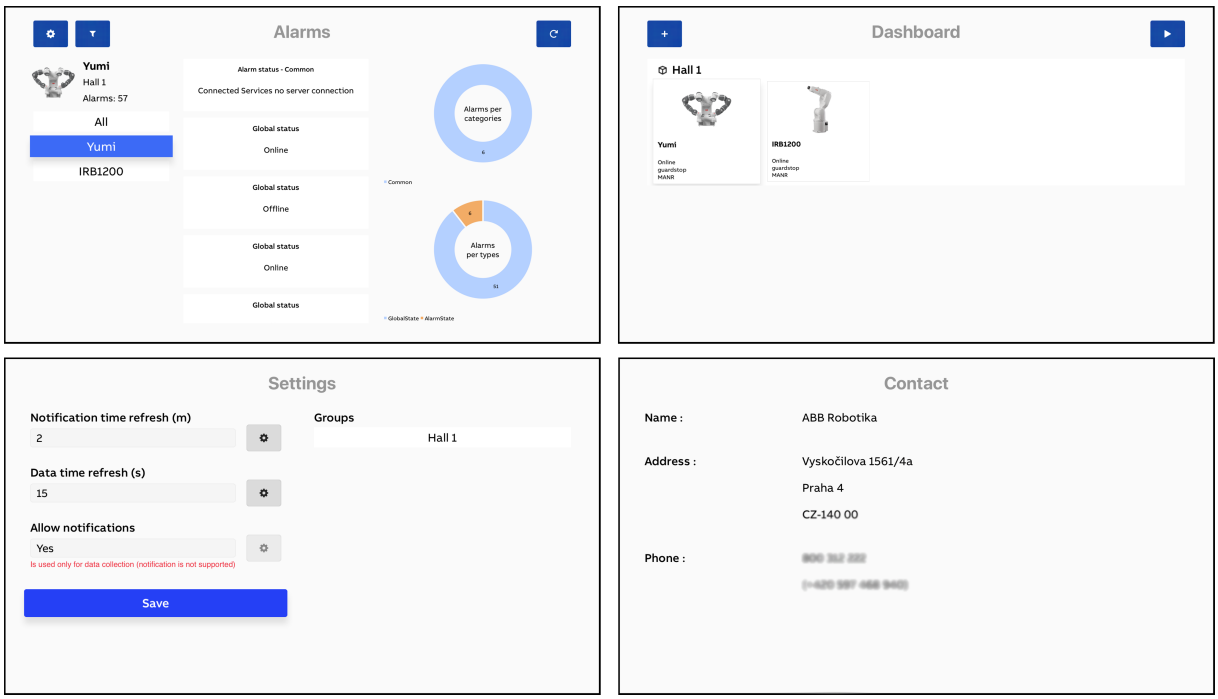
Na Obrázku 8 môžete vidieť ukážku mobilnej aplikácie.



Obr. 8: Ukážka iOS aplikácie Robot Web Service

1.2.3 tvOS

Táto aplikácia je určená pre zariadenia Apple TV a je napísaná v jazyku Swift. Ponúka rovnaké funkcie ako iOS aplikácia, avšak nebolo možné využiť webovú stránku pre zobrazovanie detailov robota a daná funkcionalita bola naimplementovaná natívne. Ukážka je zobrazená na Obrázku 9.



Obr. 9: Ukážka tvOS aplikácie Robot Web Service

1.3 Nedostatky súčasného riešenia

Aktuálne riešenie sa potýka s niekoľkými nedostatkami, medzi ktoré patria:

1.3.1 Zabezpečenie

Webová verzia aplikácie používa pre svoje zabezpečenie naviazanie na IP adresu robota a následnú obfuskáciu[6] kódu. Toto riešenie však nezabraňuje skopírovať zdrojové kódy, ich následnú analýzu a prípadné použitie v ďalších robotoch. Aplikácie pre iOS a tvOS sú dostupné v App Store[7], čo umožňuje komukoľvek ich stiahnutie a využitie pre monitorovanie robotov.

1.3.2 Rozdielne verzie API rozhraní

Pravidelne dochádza k vydaniam nových verzií RobotWare a s tým sú spojené rôzne zmeny Robot Web Services rozhrania[8]. Aktuálne riešenie si vyžaduje úpravy vo všetkých typoch aplikácií, pokiaľ dôjde k zmene rozhrania, ktoré by ovplyvnilo ich funkčnosť (zmena autentifikácie, služieb a iné).

1.3.3 Prerušenie získavania dát

Pri starších verziách RobotWare dochádza pri častom dotazovaní k prechodu robota do núdzového režimu a je nevyhnutný jeho reštart. Od verzie RobotWare 6.07 je táto chyba opravená a pri častých dotazoch vráti robot odpoveď 503 (služba nedostupná). Oba prípady však vedú k pozastaveniu získavania dát z robota vo všetkých typoch aplikácií.

1.3.4 Ukladanie dát

Aktuálne riešenie neponúka centrálne ukladanie informácií o robotoch, ktoré by bolo možné využiť pre pokročilú analýzu a reporty. Pri iOS a tvOS aplikáciach síce dochádza k lokálnemu ukladaniu dát, avšak aplikácie sú plne odkázané na priame pripojenie na robota. V prípade behu aplikácií na pozadí môže správca úloh pozastaviť dané aplikácie na základe vyťaženia operačného systému, čo má za následok prerušenie sťahovania dát.

[6] Prevedenie zdrojového kódu do takej podoby, aby ho človek nedokázal ľahko pochopiť.

[7] App Store predstavuje platformu pre digitálnu distribúciu aplikácií od spoločnosti Apple.

[8] Zoznam zmien medzi verziou 1.0 a 2.0: `https://robotapps.blob.core.windows.net/apps/RWS_BreakingChanges.pdf`

2 Návrh webového servera

Táto kapitola popisuje návrh riešenia pre optimalizáciu zberu dát z priemyselných robotov. Ako prvý je vykonaný prieskum systémov pre monitorovanie robotov. Nasleduje vízia, špecifikácia požiadaviek, dizajnové obmedzenia a samotný návrh softvéru z viacerých pohľadov. [4]

2.1 Podobné systémy

Spoločnosť ABB ako aj jej konkurent v oblasti robotiky KUKA, majú vo svojom portfóliu cloudové riešenie pre vzdialené monitorovanie robotov.

Existujúci softvér od spoločnosti ABB narozdiel od navrhovaného systému neumožňuje monitorovať robotov, ktorí nie sú dostupní prostredníctvom internetu. Jedná sa o veľmi podstatný rozdiel, pretože veľké množstvo firiem nechce aby boli ich roboti dostupní mimo podnikovú sieť. Ďalší rozdiel predstavuje nekompatibilita s existujúcimi aplikáciami Robot Web Service (Sekcia 1.2) a zacielenie na oblasť servisu a s tým spojené informácie.

2.1.1 ABB Ability^{TM} Connected Services

Jedná sa o nástroj, ktorý umožňuje vzdialené monitorovanie robotov prostredníctvom webového rozhrania s názvom *MyABB*. [5] Connected Services tvoria nasledujúce služby:

Monitorovanie stavu a diagnostika
Táto služba pomáha zaistiť rýchlejší reakčný čas, vyššiu účinnosť a vylepšenú technickú podporu. Identifikuje robotické systémy, ktoré nie sú optimálne a poskytuje informácie o potenciálnych problémoch pre dosiahnutie rýchlejšieho riešenia problémov. Medzi základné vlastnosti patria:

- Monitorovanie stavu a analýza mechanických jednotiek.
- Prehľad alarmov a diagnostika.
- Upozornenia na alarmy prostredníctvom emailu alebo SMS.
- Vytvorenie snímky systému v prípade alarmu.

Fleet Assessment
Jedná sa o analýzu, ktorá identifikuje robotov, komponenty a softvér, ktoré nebežia optimálne a poskytuje odporúčania pre vylepšenie ich výkonu a životnosti.

Asset Optimization
Táto služba porovnáva jednotlivých robotov užívateľa s celou populáciou robotov, ktorá je pripojená do platformy Connected Services a na základe výsledkov uprednostňuje servisné úkony.

Vzdialený prístup
Vzdialený prístup umožňuje riadenie robota a pripojených zariadení z bezpečného miesta mimo dosah fyzického ovládacieho panela.

Správa záloh
Umožňuje automatické alebo manuálne vytváranie záloh a poskytuje ich historický prehľad. Tieto zálohy môžu byť ukladané do cloudu alebo na ľubovoľný server.

2.1.2 KUKA Connect

KUKA Connect predstavuje platformu, ktorá umožňuje sťahovať dáta z robotov s ich následnou analýzou a vizualizáciou. Je k dispozícií vo forme webu alebo mobilnej aplikácie pre Android a iOS zariadenia. [6] [7] Platforma poskytuje nasledujúcu funkcionalitu:

Administrácia
Táto sekcia je určená pre administrátorov a nie je dostupná pre mobilné zariadenia. Administrácia je rozdelená do troch častí:

- **Správa užívateľov:** Umožňuje pridávanie užívateľov, správu rolí a priraďovanie robotov.
- **Správa robotov:** Roboti sú automaticky priradení do geografickej oblasti, v ktorej sa nachádzajú. Týchto robotov je ďalej možné manuálne triediť do skupín a tímov.
- **Správa alarmov:** Táto časť umožňuje nastaviť, ktoré alarmy sa budú zobrazovať užívateľom.

Prehľad robotov
Jedná sa o domovskú stránku aplikácie, ktorá zobrazuje všetkých robotov priradených k užívateľovi a má nasledujúce vlastnosti:

- **Filtrovanie robotov:** Je možné filtrovať robotov na základe rôznych parametrov ako organizačná hierarchia, verzia softvéru alebo konfigurácia.
- **Detail robota:** Po vybratí konkrétneho robota sa zobrazia jeho detaily (status, IP adresa, doba prevádzky, verzia softvéru). Ďalej je možné zobraziť 3D vizualizáciu robota, ktorá zachytáva jeho pozíciu a pre niektoré modely aj teploty na jednotlivých osiach.

Údržba
Táto sekcia automatizuje plánovanie rutinných údržbárskych úloh. Umožňuje sledovať minulé, súčasné a budúce úkony údržby pre jednotlivých robotov (základná údržba, zálohovanie dát, kontrola hlavnej osi, ...).

Správy a alarmy
Pokiaľ nastane chyba na niektorom z priradených robotov, dôjde k zobrazeniu upozornenia a prípadnému odoslaniu SMS správy. Užívateľ má k dispozícií detailný náhľad na jednotlivé alarmy a taktiež sa tu nachádza široká škála grafov (roboti s najväčším počtom alarmov, alarmy s najčastejším výskytom, ...).

Sledovanie stavu
Slúži na zobrazenie kľúčových informácií o stave robotov (vyťaženie procesora, množstvo miesta na disku alebo stav batérie). Ďalej táto časť poskytuje historickú analýzu, ktorá zobrazuje normálny rozsah prevádzkového stavu motora robota v čase a vykresľuje dátové body mimo normu.

Protokol zmien
Všetky informácie týkajúce sa zmien hardvéru, softvéru a konfigurácie robota.

AGV monitorovanie
Táto sekcia je určená pre zobrazovanie informácií pre autonómnych mobilných robotov KUKA a je organizovaná do troch častí:

- **Stav:** Informácie o stave robota.
- **Mapa:** 2D prostredie pohybu robota.
- **Grafy:** Vizualizácia monitorovaných dát.

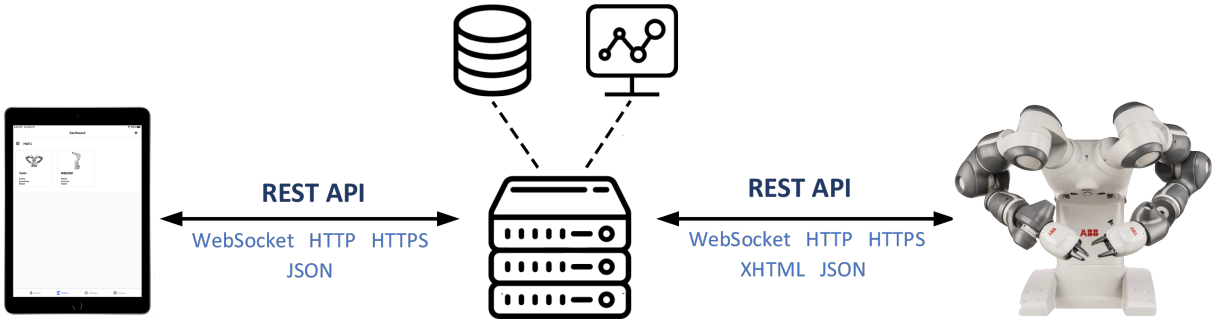
2.2 Vízia

Chceme vytvoriť webový server, ktorý umožní zber dát z priemyselných robotov, zobrazí ich užívateľovi a poskytne v jednotnom formáte jednotlivým klientom. Princíp fungovania je zobrazený na Obrázku 10. Webový server zabezpečí nasledujúcu funkcionalitu:

- **Nepretržitý zber dát**
- **Zjednotenie verzií Robot Web Services**
- **WebSocket komunikácia**
- **Zabezpečenie existujúcich aplikácií**
- **Automatické vyhľadávanie robotov**
- **Centrálny prístupový bod pre všetky zariadenia**

Role užívateľov, ktorí budú s aplikáciou pracovať sú:

- **Užívateľ:** Zobrazovanie získavaných dát, priebehu komunikácie, robotov a ich detailov.
- **Administrátor:** Operácie spojené s editáciou robotov, užívateľov, skupín a koncových bodov.
- **Zariadenie:** Získavanie dát prostredníctvom REST API rozhrania pre potreby iOS a tvOS aplikácií.



Obr. 10: Robot Web Service server komunikácia

2.3 Špecifikácia požiadaviek

Nasledujúca špecifikácia vychádza zo zadania diplomovej práce a taktiež z existujúcich požiadaviek pre projekt Robot Web Service.

2.3.1 Koncové body

Aplikácia umožní pridávanie jednotlivých služieb z rozhrania Robot Web Services, popísaných v Sekcii 1.1.2. Užívateľ si bude schopný uložiť službu pod vlastným názvom a tá bude jedinečne identifikovaná pomocou URL adresy. Koncový bod bude ďalej obsahovať typ (*Http, WebSocket*) a prioritu (*Nízka, Stredná, Vysoká*). Koncovým bodom bude možné priradiť parametre na úpravu dotazu a taktiež parsre pre získanie dát z odpovede.

2.3.2 Roboti

Užívateľ bude môcť pridávať robotov automaticky alebo manuálne. Automatické pridávanie zobrazí robotov v sieti s predvyplnenou IP adresou. V manuálnom režime bude musieť IP adresu zadať užívateľ. Ďalej bude potrebné doplniť názov, prihlasovacie údaje, typ autentifikácie (*Basic, Digest*), sériové číslo a voľbu pre získavanie dát (*Http, WebSocket, Oboje, Žiadne*). Pre začatie komunikácie bude potrebné každému robotovi nadefinovať, ktoré koncové body má využívať. Súčasťou bude aj otestovanie dostupnosti robota.

2.3.3 Skupiny

Skupiny budú slúžiť na organizovanie robotov, pričom jedného robota bude možné priradiť do viacerých skupín. Každá skupina bude obsahovať jedinečný názov.

2.3.4 Užívatelia

Táto sekcia bude slúžiť na vytváranie užívateľov pre webový server ako aj mobilných klientov. Užívateľom webovej aplikácie bude možné priradiť rolu (*Užívateľ, Administrátor*) a tým im obmedziť prístup do jednotlivých sekcií. Užívatelia pre iOS a tvOS aplikácie budú naviazaní na konkrétne zariadenie. Poslednou funkcionalitou bude možnosť priradenia robotov jednotlivým užívateľom.

2.3.5 Dashboard

Táto časť ponúkne prehľad o všetkých robotoch priradených k užívateľovi. Budú sa tu nachádzať informácie o stave pripojenia a užívateľ si bude schopný vybrať robota a zobraziť jeho detaily. Robotov bude možné filtrovať na základe skupín.

2.3.6 Detail robota

Detail robota poskytne všetky informácie, ktoré sú aktuálne dostupné vo webovej verzií Robot Web Service, viď Sekcia 1.2.1.

2.3.7 Alarmy

Táto sekcia poskytne prehľad alarmov, ktoré nastali na jednotlivých robotoch. Alarmy bude možné filtrovať na základe dátumu, kategórií alebo robotov.

2.3.8 Dáta

V tejto časti užívateľ uvidí dáta získané z robotov pre nadefinované koncové body. Podobne ako v prípade alarmov, bude možné vykonávať filtrovanie získaných dát (dátum, koncové body, roboti).

2.3.9 Komunikácia

Táto časť bude slúžiť na prehľad komunikácie servera s robotmi. Užívateľ bude mať k dispozícií informácie o začatí komunikácie, jej priebehu ako aj prípadných výpadkoch. Dáta bude možné filtrovať podľa dátumu, robotov a typov udalostí.

2.4 Dizajnové a implementačné obmedzenia

Jedná sa o obmedzenia, ktoré sú uvalené na konštrukčné riešenie informačného systému. Môžu sa týkať softvéru, hardvéru alebo ktorejkoľvek časti systému. Tieto obmedzenia zvyčajne ukladá zákazník alebo organizácia zodpovedná za vývoj.

2.4.1 Server

Webový server bude napísaný v jazyku C#. Konkrétne sa bude jednať o *ASP.NET Core* aplikáciu vo verzii 3.1, ktorá bude hosťovaná v Docker kontajneri. Server bude komunikovať s klientmi prostredníctvom REST API rozhrania s využitím HTTP a WebSocket protokolu.

2.4.2 Klient

Klientská časť aplikácie bude implementovaná v JavaScriptovom frameworku *React* s využitím stavového kontajnera *Redux*. Ako programovací jazyk bude použitý *TypeScript*, ktorý rozširuje JavaScript o statickú typovú kontrolu. Vzhľadom na firemnú politiku bude klient optimalizovaný pre prehliadače Google Chrome a Microsoft Edge (stará aj nová verzia). Podobne ako server, aj klient bude hosťovaný v Docker kontajneri.

2.4.3 Bezpečnosť

Vytváranie užívateľských účtov pre webovú časť aplikácie a mobilných klientov bude vykonávať administrátor. Všetky heslá budú šifrované. Po úspešnom autentifikovaní obdrží klient od servera *JWT* token, ktorý bude využívaný pre nasledovnú komunikáciu. Exspirovaný token bude možné obnoviť v definovanom časovom období bez nutnosti opätovného zadávania prihlasovacích údajov pomocou *refresh* tokenu. Prístupové údaje pre mobilných klientov budú navyše spárované s konkrétnym zariadením.

2.4.4 Dátová perzistencia

Pre trvalé uloženie dát bude slúžiť relačná databáza *PostgreSQL*, ktorá bude hosťovaná v Docker kontajneri. Roboti, skupiny, užívatelia a koncové body nebudú fyzicky mazané ale budú obsahovať príznak o zmazaní, aby bolo možné dohľadať historické dáta.

2.4.5 Testovanie

Systém bude podrobený viacerým typom testovania, aby bola zaistená vysoká spoľahlivosť pred samotným nasadením do produkcie. Bude sa jednať o nasledujúce spôsoby testovania:

- **Unit testovanie**
- **Výkonnostné testovanie**
- **Statická analýza kódu**

2.4.6 Hardvér

Na základe požiadaviek od zadávateľa projektu (malé rozmery, nízka cena), bude riešenie postavené na zariadení Raspberry Pi 3B+[9]. Tieto mini počítače disponujú širokou komunitou vývojárov, umožňujú nainštalovať rozličné operačné systémy (*Raspbian, Ubuntu Server, Windows 10 IoT, ...*) a v neposlednom rade podporujú Docker virtualizáciu. Základné technické parametre sú uvedené v Tabuľke 2.

Tabuľka 2: Technická špecifikácia Raspberry Pi 3B+

CPU	1.4 GHz 64-bit quad-core ARM Cortex-A53
RAM	1 GB
Pamäť	Micro SD karta
Sieť	Gigabit Ethernet skrz USB 2.0 (max 300 Mb/s)
WiFi	2.4GHz & 5GHz 802.11.b/g/n/ac

2.4.7 Vývojové nástroje

Pri vývoji budú využité nasledujúce nástroje:

- **Visual Studio Professional 2019:** Implementácia serverovej časti aplikácie.
- **Visual Studio Code:** Implementácia klientskej časti aplikácie.
- **PgAdmin 4:** Práca s databázou.
- **Enterprise Architect 15:** Kreslenie diagramov.
- **Azure DevOps:** Verziovanie kódu, zostavenie a nasadenie aplikácie.

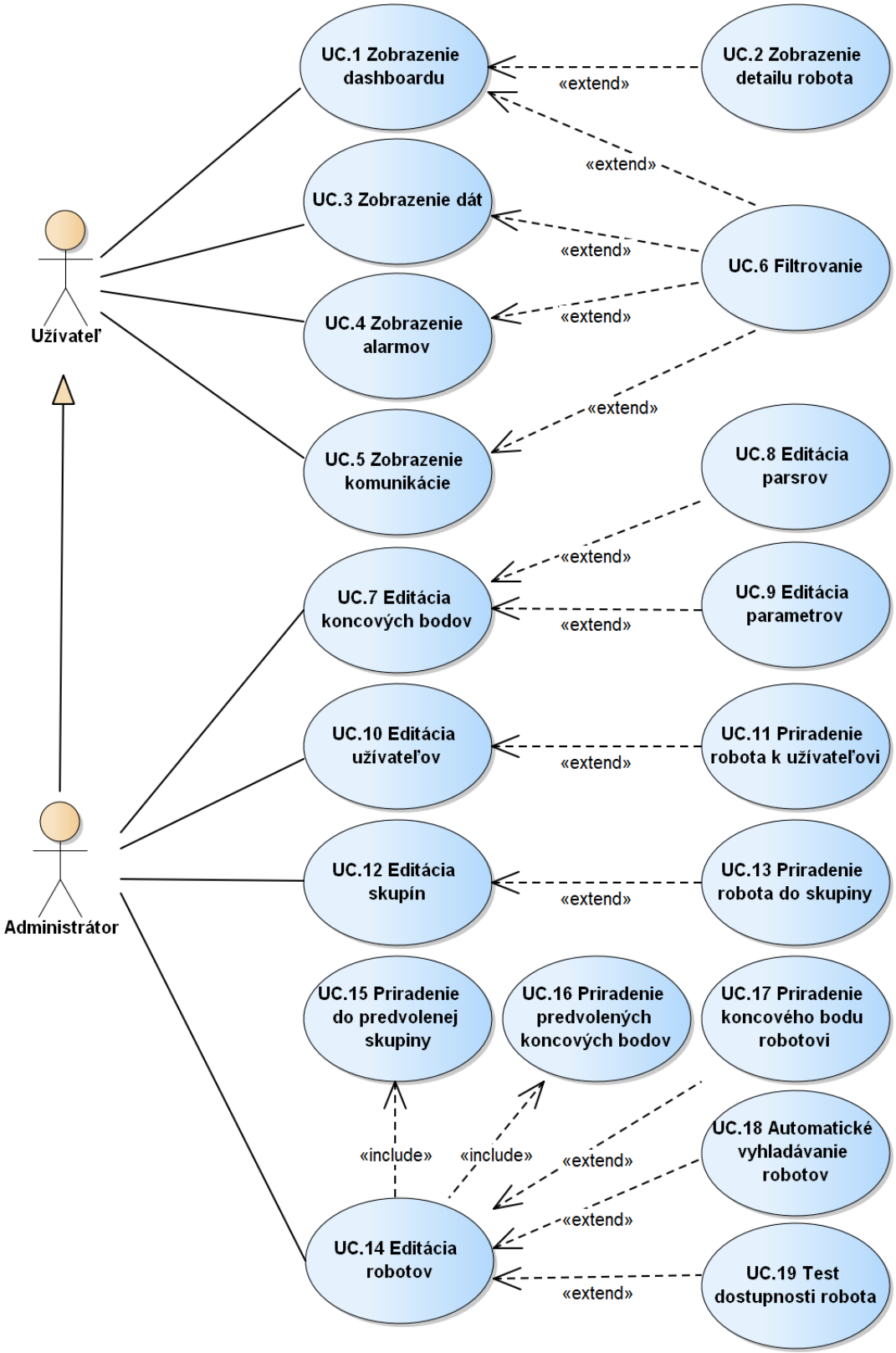
2.5 Use Case pohľad

Slúži na zachytenie funkčnosti systému z pohľadu vonkajšieho sveta. Na Obrázku 11 sú zobrazené jednotlivé prípady užitia spolu s aktérmi a ich vzájomnými vzťahmi.

2.5.1 Use Case popis

Táto časť podrobne popisuje prípady užitia, ktoré majú najväčší dopad na architektúru systému. Prerekvizitou pre všetky prípady užitia je prihlásenie do systému.

[9] `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus`



Obr. 11: Use Case Diagram

UC.7 Editácia koncových bodov
Editácia koncových bodov umožní pridanie, aktualizáciu a odobranie dostupných služieb robota. Koncovým bodom bude ďalej možné pridať parsre, pomocou ktorých sa vyberú dáta z odpovede, poprípade parametre pre úpravu dotazu.

Aktéri: Administrátor
Spúšťač: Užívateľ chce spravovať koncové body
Podmienky: Prihlásenie do systému
Exclude: UC.8 Editácia parsrov, UC.9 Editácia parametrov

Pridanie koncového bodu

Hlavný scenár:

1. Systém: poskytne formulár pre pridanie koncového bodu
2. Aktér: vyplní formulár
3. Systém: skontroluje zadané údaje
4. Systém: uloží koncový bod do databázy
5. Systém: notifikuje užívateľa

Vedľajší scenár:

3.1 Systém: vyzve na opätovné vyplnenie údajov

3.2 Systém: pokračuje hlavným scenárom, bod č.2

Aktualizácia koncového bodu

Hlavný scenár:

1. Systém: poskytne zoznam koncových bodov
2. Aktér: vyberie koncový bod
3. Systém: poskytne formulár pre aktualizáciu koncového bodu
4. Aktér: upraví údaje o koncovom bode
5. Systém: skontroluje zadané údaje
6. Systém: uloží zmeny do databázy
7. Systém: notifikuje užívateľa
8. Systém: zaktualizuje komunikáciu s robotmi, ktorí využívajú daný koncový bod

Vedľajší scenár:

5.1 Systém: vyzve na opätovné vyplnenie údajov

5.2 Systém: pokračuje hlavným scenárom, bod č.4

Odstránenie koncového bodu

Hlavný scenár:

1. Systém: poskytne zoznam koncových bodov
2. Aktér: vyberie koncový bod
3. Systém: zobrazí výzvu pre odstránenie
4. Aktér: potvrdí žiadosť o odstránenia
5. Systém: zmaže koncový bod, parsre a parametre z databázy
6. Systém: notifikuje užívateľa
7. Systém: zaktualizuje komunikáciu s robotmi, ktorí využívajú daný koncový bod

Vedľajší scenár:

4.1 Aktér: zruší žiadosť o odstránenie

4.2 Systém: pokračuje hlavným scenárom, bod č.1

UC.14 Editácia robotov
Tento prípad užitia umožní pridanie, aktualizáciu alebo odstránenie robotov. Súčasťou bude aj automatické vyhľadávanie robotov v sieti a testovanie ich dostupnosti. Jednotlivým robotom bude možné priraďovať koncové body.

Aktéri: Administrátor
Spúšťač: Užívateľ chce spravovať robotov
Podmienky: Prihlásenie do systému
Include: UC.15 Priradenie do predvolenej skupiny, UC.16 Priradenie predvolených koncových bodov
Exclude: UC.17 Priradenie koncového bodu robotovi, UC.18 Automatické vyhľadávanie robotov, UC.19 Test dostupnosti robota

Pridanie robota

Hlavný scenár:

1. Systém: poskytne formulár pre pridanie robota
2. Aktér: vyberie robota zo zoznamu vyhľadaných robotov «exclude» UC.18
3. Aktér: vyplní formulár
4. Aktér: otestuje dostupnosť robota «exclude» UC.19
5. Systém: skontroluje zadané údaje
6. Systém: uloží robota do databázy
7. Systém: priradí robota do predvolenej skupiny «include» UC.15
8. Systém: priradí predvolené koncové body robotovi «include» UC.16
9. Systém: notifikuje užívateľa
10. Systém: spustí komunikáciu s robotom

Vedľajší scenár:

5.1 Systém: vyzve na opätovné vyplnenie údajov

5.2 Systém: pokračuje hlavným scenárom, bod č.3

Aktualizácia robota

Hlavný scenár:

1. Systém: poskytne zoznam robotov
2. Aktér: vyberie robota
3. Systém: poskytne formulár pre aktualizáciu robota
4. Aktér: zaktualizuje údaje robota
5. Aktér: otestuje dostupnosť robota «exclude» UC.19
6. Systém: skontroluje zadané údaje
7. Systém: uloží zmeny do databázy
8. Systém: notifikuje užívateľa
9. Systém: zaktualizuje komunikáciu s daným robotom

Vedľajší scenár:

4.1 Aktér: vyberie robota zo zoznamu vyhľadaných robot «exclude» UC.18

4.2 Systém: pokračuje hlavným scenárom, bod č.4

6.1 Systém: vyzve na opätovné vyplnenie údajov

6.2 Systém: pokračuje hlavným scenárom, bod č.4

Odstránenie robota

Hlavný scenár:

1. Systém: poskytne zoznam robotov
2. Aktér: vyberie robota
3. Systém: zobrazí výzvu pre odstránenie
4. Aktér: potvrdí žiadosť o odstránenie
5. Systém: zmaže robota z databázy
6. Systém: notifikuje užívateľa
7. Systém: ukončí komunikáciu s daným robotom

Vedľajší scenár:

4.1 Aktér: zruší žiadosť o odstránenie

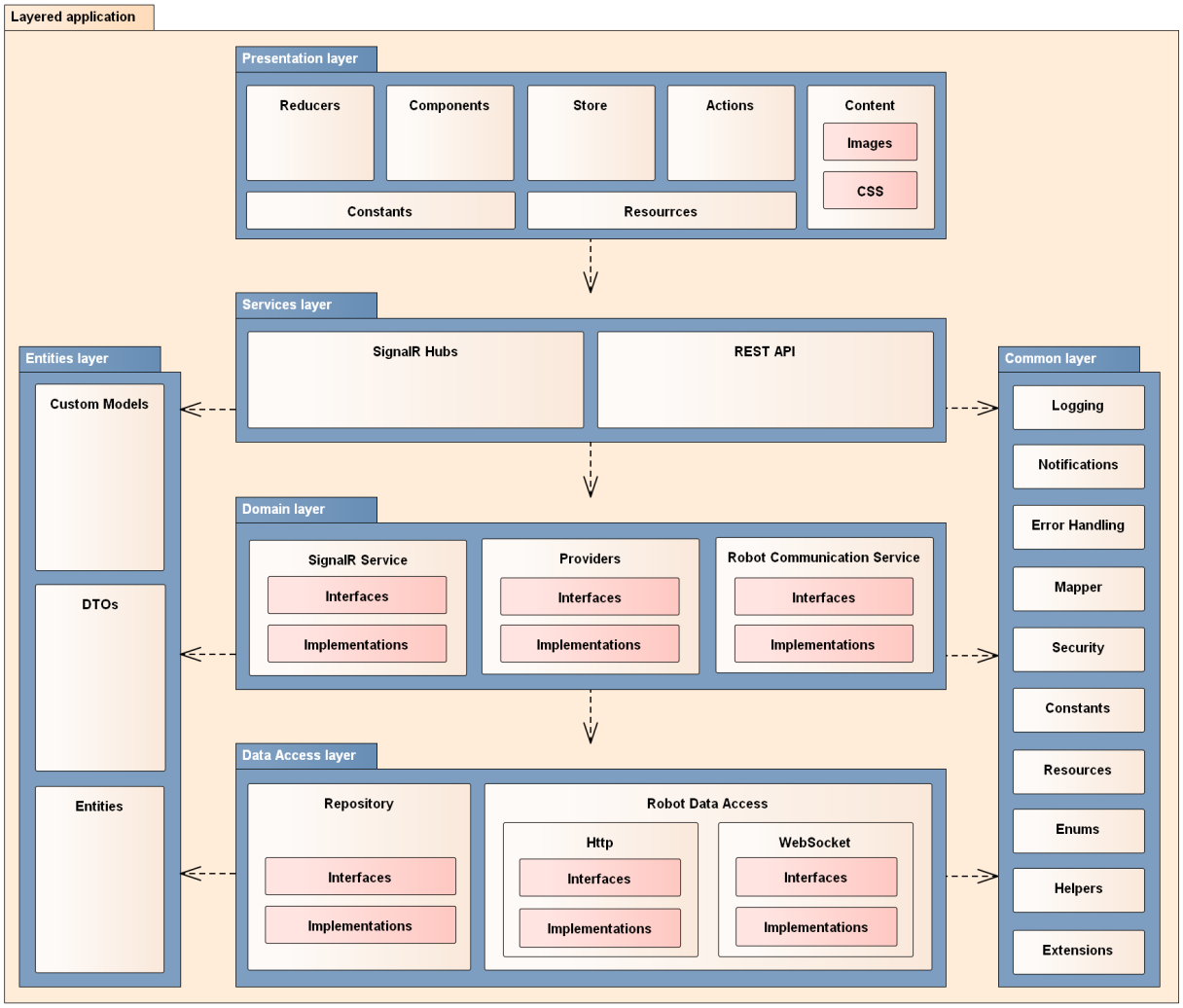
4.2 Systém: pokračuje hlavným scenárom, bod č.1

2.6 Implementačný pohľad

Implementačný pohľad slúži na popis systému z pohľadu programátora. Zameriava sa na organizáciu kódu, jeho hlavných modulov a ich závislostí.

2.6.1 Návrh architektúry

Systém z pohľadu architektúry bude rozdelený do viacerých vrstiev. Toto rozdelenie je zobrazené na Obrázku 12.



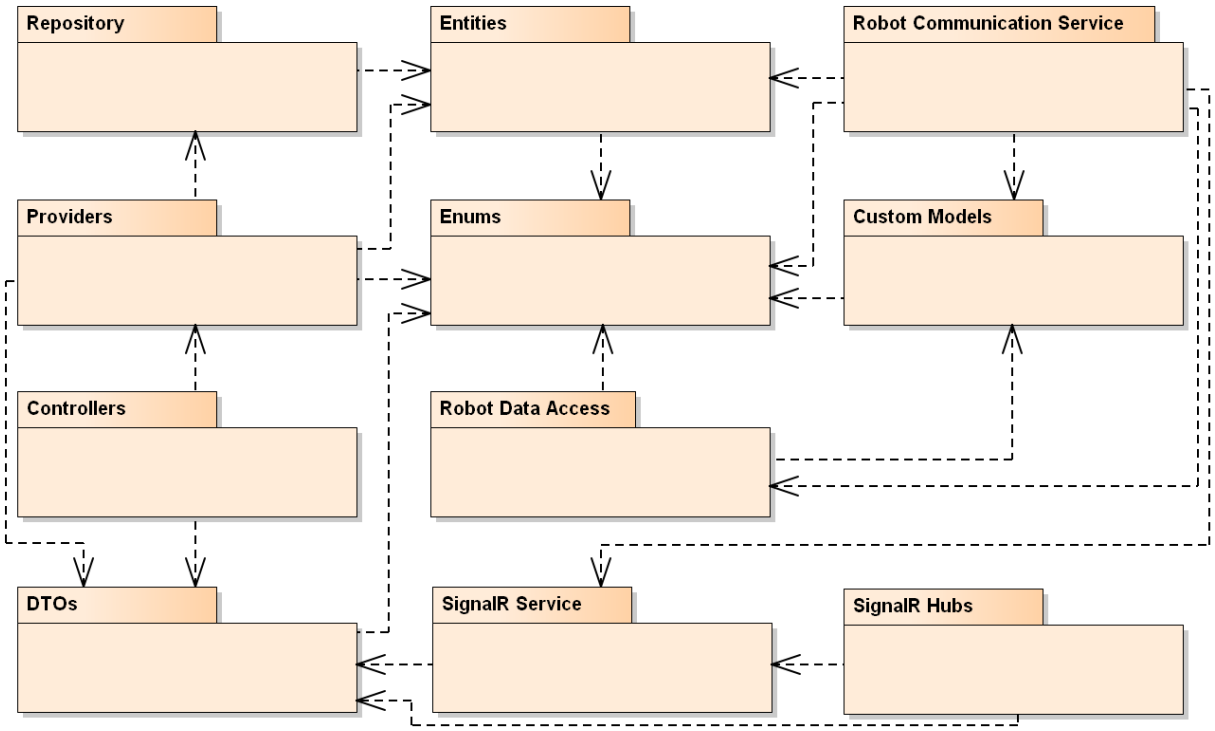
Obr. 12: Diagram návrhu architektúry

Zoznam vrstiev spolu so stručným popisom:

- **Data Access layer:** Prístup k databáze a získanie dát z robota.
- **Domain layer:** Spracovanie dát a operácie nad nimi.
- **Service layer:** Poskytnutie dát jednotlivým klientom.
- **Entities layer:** Dátové objekty pre jednotlivé vrstvy.
- **Common layer:** Podporné funkcie zdieľané viacerými vrstvami.
- **Presentation layer:** Vizualizácia získaných dát a interakcia s užívateľom.

2.6.2 Diagram balíčkov

Obrázok 13 zachytáva závislosti medzi najdôležitejšími balíčkami v systéme.



Obr. 13: Diagram balíčkov

2.7 Logický pohľad

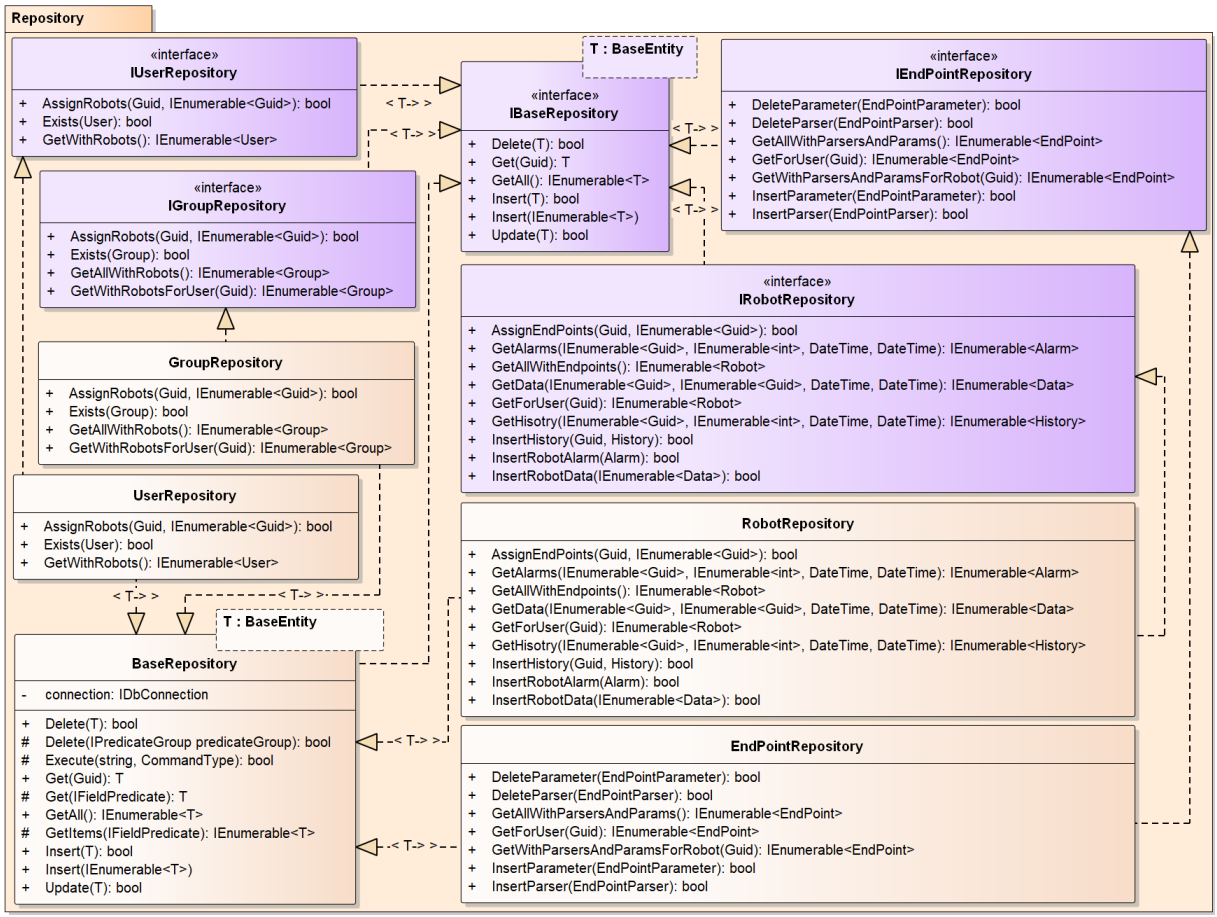
Logický pohľad slúži na popísanie funkcionality systému z hľadiska jeho statickej štruktúry.

2.7.1 Triedny diagram

V tejto časti sú vybrané balíčky zo Sekcie 2.6.2 podrobnejšie popísané z pohľadu tried a ich vzájomných vzťahov.

Repository

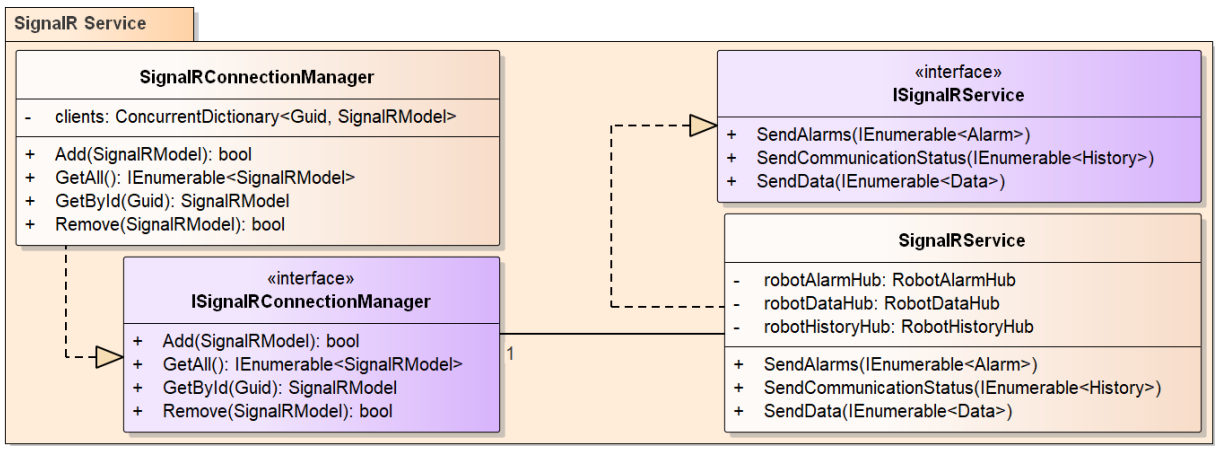
Triedny diagram generického repozitára je zobrazený na Obrázku 14. Toto riešenie umožňuje v prípade potreby rýchlo a jednoducho rozšíriť repozitár o novú triedu. Stačí implementovať rozhranie *IBaseRepository* a dediť od triedy *BaseRepository* - to umožní triede využiť všetky existujúce CRUD operácie bez nutnosti ich implementácie.



Obr. 14: Triedny diagram - Repository

SignalR Service

Na Obrázku 15 sú zobrazené triedy, ktoré zaobstarávajú komunikáciu s jednotlivými klientmi pomocou SignalR protokolu.



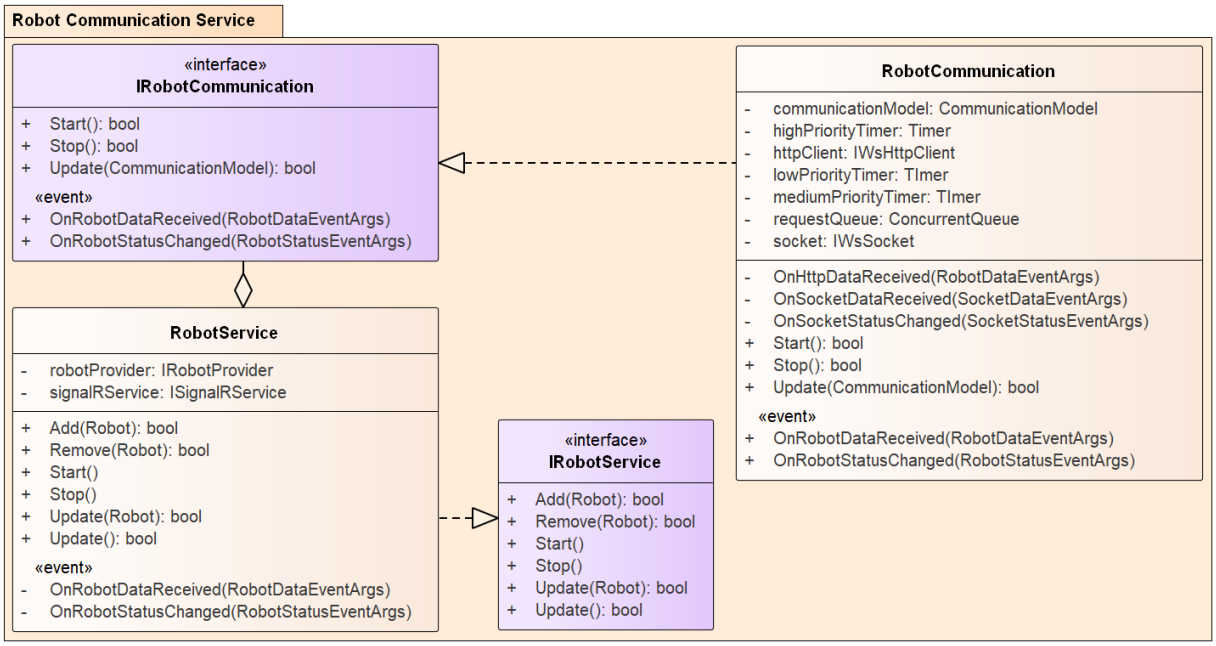
Obr. 15: Triedny diagram - SignalR Service

Stručný popis jednotlivých tried:

- **SignalRConnectionManager:** Stará sa o pridávanie alebo odoberanie klientov a udržuje si zoznam aktívnych spojení.
- **SignalRService:** Zabezpečuje odoslanie dát pripojeným klientom.

Robot Communication Service

Tieto triedy slúžia na nadviazanie, udržanie, aktualizovanie alebo prerušenie komunikácie s robotom, viď Obrázok 16.



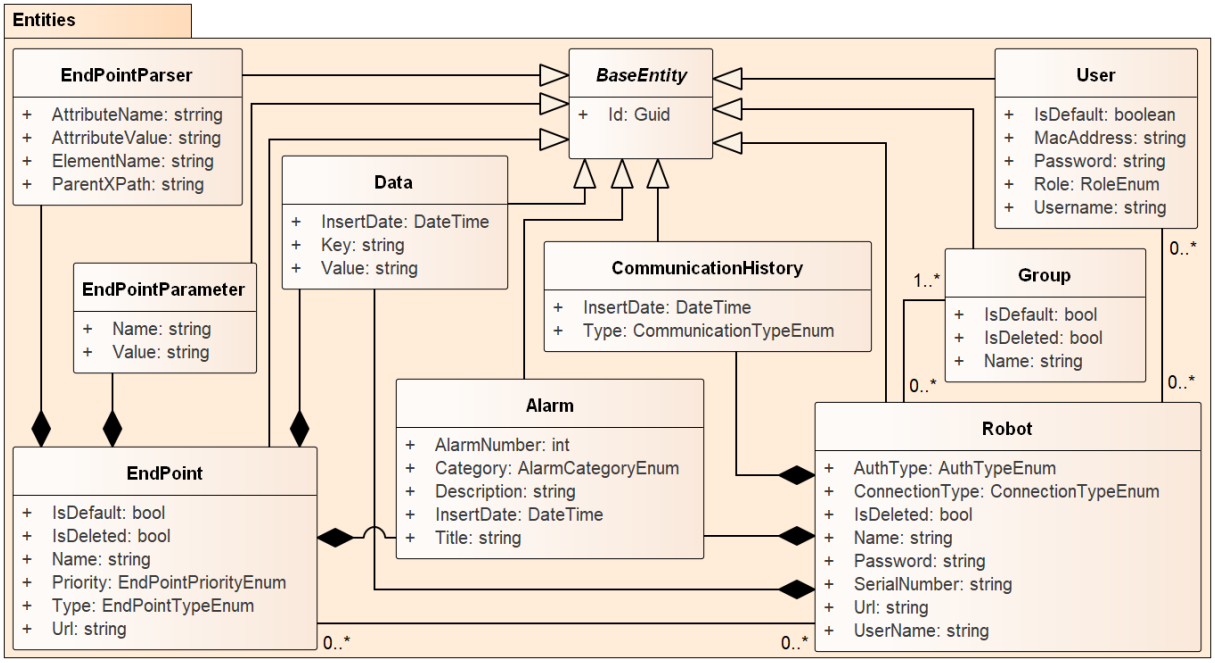
Obr. 16: Triedny diagram - Robot Communication Service

Stručný popis jednotlivých tried:

- **RobotCommunication:** Zabezpečuje komunikáciu s robotom. V pravidelných intervaloch odosiela požiadavky na server, parsruje odpovede a predáva ich triede *RobotService*. Rovnakým spôsobom spracováva dáta získané prostredníctvom WebSocket pripojenia.
- **RobotService:** Udržuje zoznam aktívnych pripojení na robotov a obsahuje operácie na ich pridanie, aktualizáciu alebo odobranie. Ďalej ukladá získané dáta a predáva ich triede *SignalRService*.

Entities

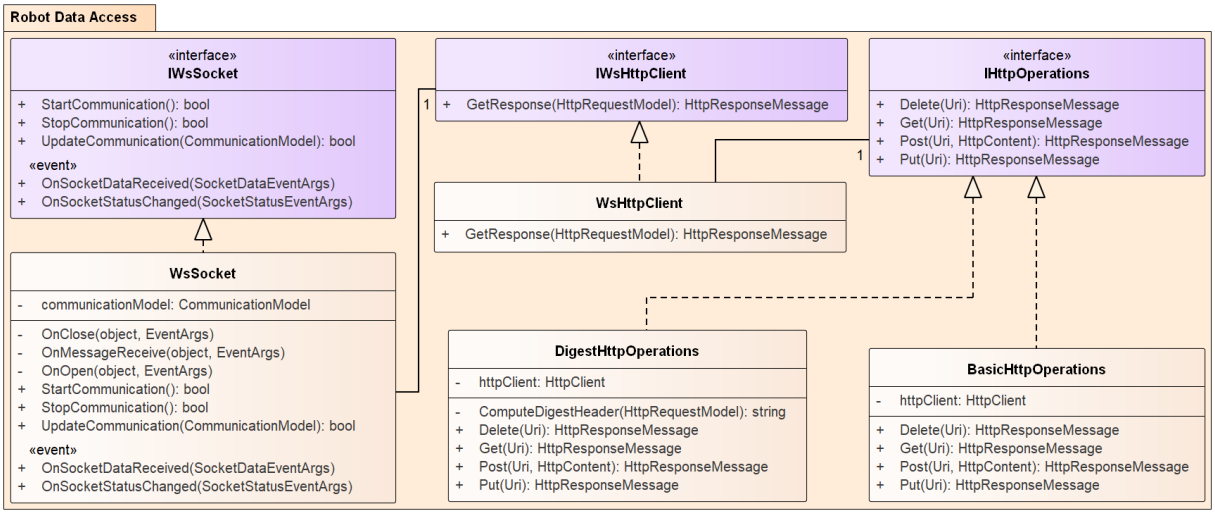
Na Obrázku 17 môžete vidieť triedny diagram entít, ktoré sú využívané doménovou vrstvou a vrstvou pre prístup k databáze.



Obr. 17: Triedny diagram - Entities

Robot Data Access

Triedy z Obrázka 18 majú za úlohu nadviazať spojenie s robotom a získať potrebné dáta. Vďaka využitiu návrhového vzoru *strategy* je možné jednoducho pridať nový typ autentifikácie pre HTTP komunikáciu.



Obr. 18: Triedny diagram - Robot Data Access

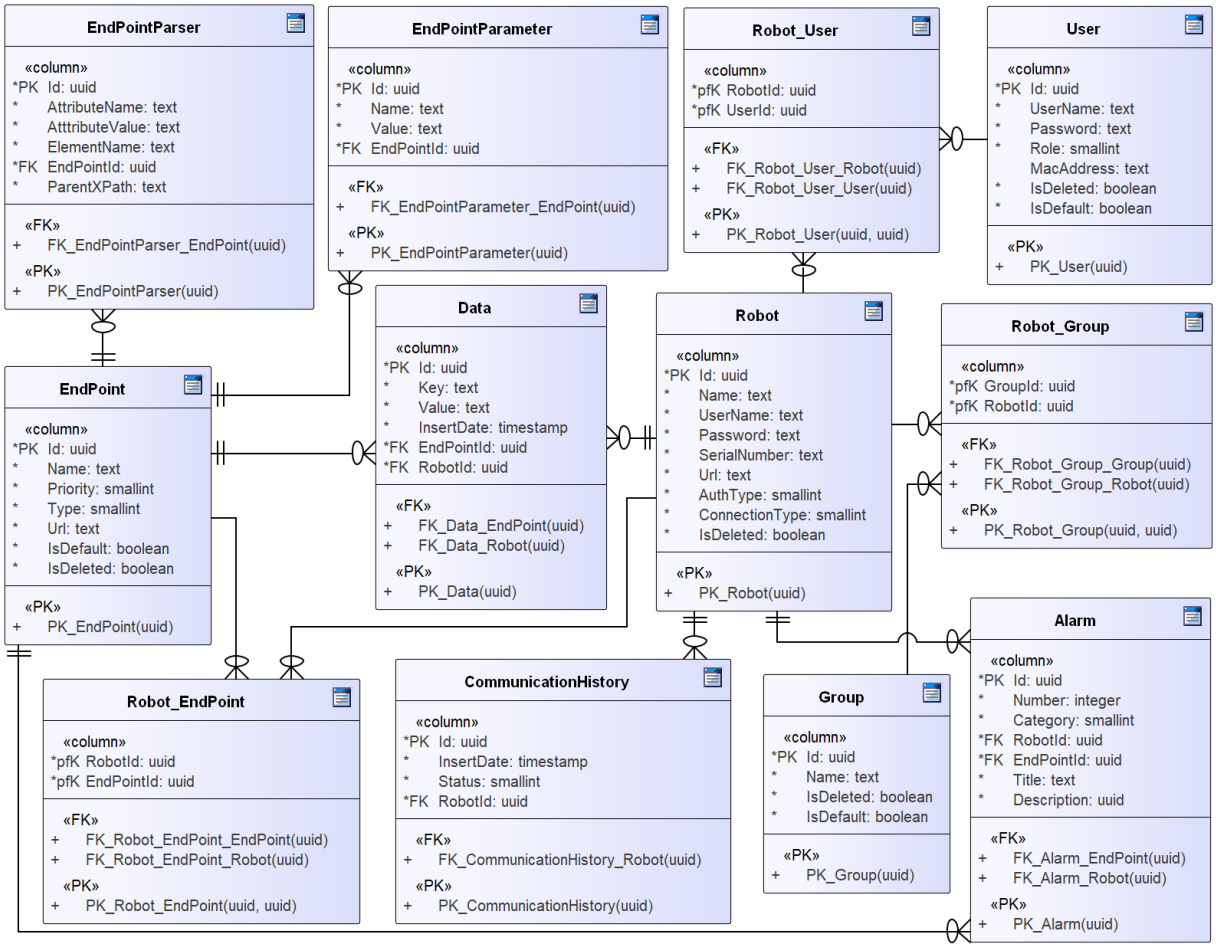
Stručný popis jednotlivých tried:

- **WsHttpClient:** Zabezpečuje komunikáciu prostredníctvom HTTP protokolu.

- **DigestHttpOperations:** HTTP operácie využívajúce Digest autentifikáciu.
- **BasicHttpOperations:** HTTP operácie využívajúce Basic autentifikáciu.
- **WsSocket:** Metódy pre začatie, aktualizáciu alebo ukončenie WebSocket pripojenia. Taktiež obsahuje udalosti, ktoré môžu nastať počas komunikácie (otvorenie, uzavretie, prijatie správy).

2.7.2 ER diagram

Obrázok 19 zobrazuje databázový model z pohľadu jednotlivých tabuliek a vzťahov medzi nimi.



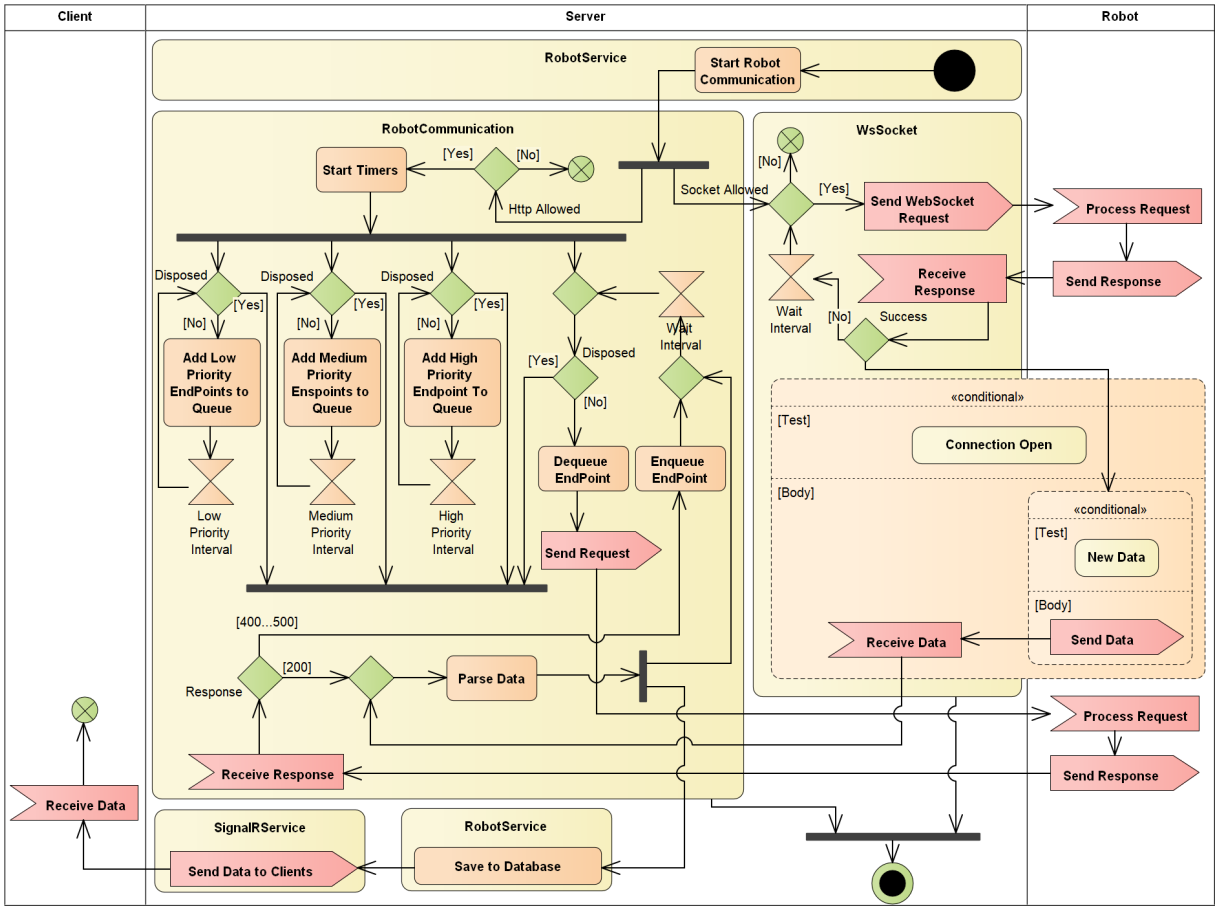
Obr. 19: ER diagram

2.8 Procesný pohľad

Procesný pohľad sa zaoberá dynamickými aspektami systému, spôsobmi komunikácie a zameriava sa na chovanie softvéru za behu.

2.8.1 Diagram aktivít

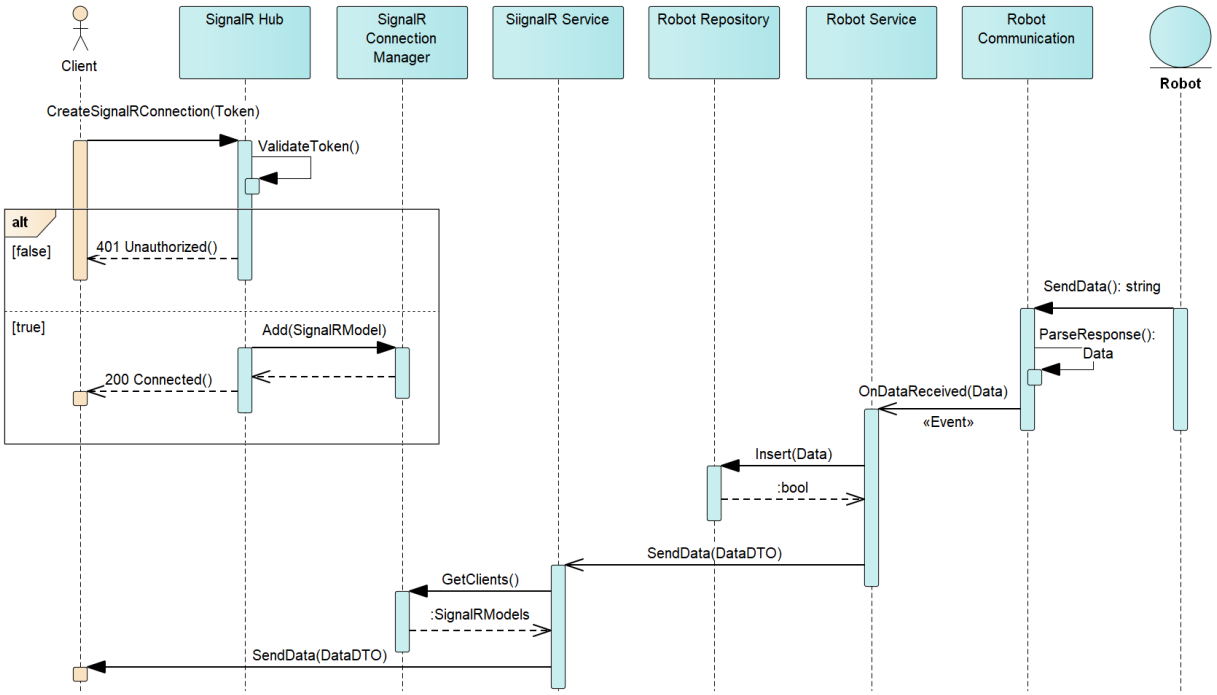
Na Obrázku 20 je zobrazený diagram aktivít popisujúci komunikáciu s robotom prostredníctvom HTTP požiadaviek a WebSocket protokolu. HTTP komunikáciu zabezpečuje časovač, ktorý v pravidelnom intervale vyberá koncové body z fronty a posiela požiadavky na robota. Ďalšie časovače slúžia na pridávanie koncových bodov do fronty. WebSocket komunikácia ako aj jednotlivé časovače bežia v samostatných vláknach. Po prijatí odpovede dôjde k jej rozparsrovaniu, uloženiu do databázy a následnému odoslaniu pripojeným klientom.



Obr. 20: Diagram aktivít pre komunikáciu s robotom

2.8.2 Sekvenčný diagram

Sekvenčný diagram na Obrázku 21 znázorňuje nadviazanie SignalR pripojenia z pohľadu klienta. Pre úspešné pripojenie je najskôr potrebné získať JWT token, ktorý je následne odoslaný s požiadavkou o nadviazanie spojenia. Diagram ďalej zobrazuje postupnosť krokov od získania dát z robota, cez ich uloženie až po odoslanie jednotlivým klientom.



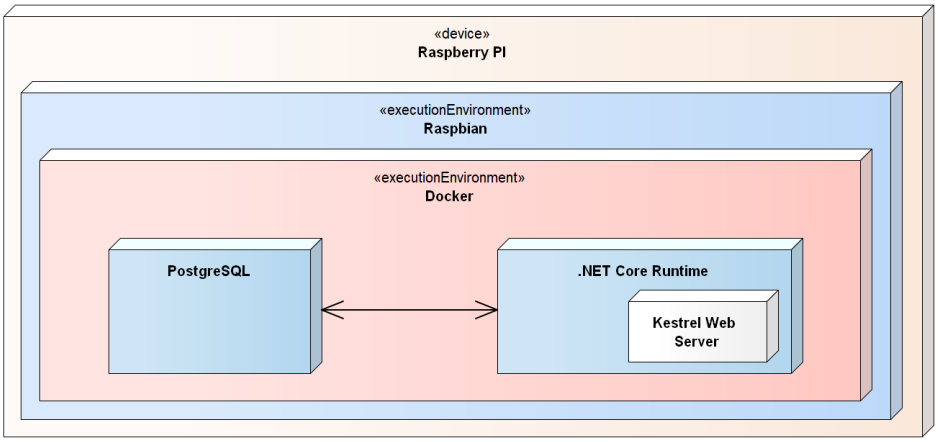
Obr. 21: Sekvenčný diagram pre SignalR komunikáciu

2.9 Pohľad nasadenia

Tento pohľad zachytáva rozmiestnenie softvérových komponent naprieč fyzickou architektúrou, ako aj komunikáciu medzi týmito komponentami.

2.9.1 Diagram nasadenia

Aplikácia bude nasadená na zariadení Raspberry Pi 3B+ s operačným systémom *Raspbian Lite*. Tento operačný systém bude ďalej obsahovať virtualizačnú platformu Docker, ktorá bude slúžiť ako behové prostredie pre kontajnery obsahujúce jednotlivé súčasti aplikácie, viď Obrázok 22.



Obr. 22: Diagram nasadenia

3 Implementácia

Táto časť diplomovej práce sa zaoberá implementáciou webového servera na základe návrhu z predchádzajúcej kapitoly, ako aj úpravou existujúcich aplikácií. Podrobne popisuje vybrané detaily implementácie, použité knižnice a vzniknuté problémy.

3.1 Server

3.1.1 Použité knižnice

Spoločnosť Microsoft umožňuje vývojárom jednoducho pridávať knižnice do svojich .NET projektov prostredníctvom správcu balíčkov s názvom *NuGet*. Všetky knižnice použité pri vývoji pochádzajú z verejného repozitára *nuget.org*. [8]

Autofac

Jedná sa o knižnicu, ktorá zabezpečuje správu závislostí medzi triedami pomocou IoC kontajnera. Tento prístup umožňuje vytvárať moduly, ktoré sú jednoducho zameniteľné a testovateľné. Princíp spočíva v tom, že IoC kontajner vytvára objekt špecifickej triedy a *injektuje* všetky závislé objekty prostredníctvom konštruktora alebo metódy a tým nás odbremeňuje od toho, aby sme vytvárali a spravovali objekty ručne. [9]

Súčasťou .NET Core frameworku je aj IoC kontajner, ktorý však neponúka také množstvo funkcií ako Autofac a kvôli špecifickým potrebám projektu bol zvolený práve tento balíček. Ukážka registrácie komponent do IoC kontajnera je zobrazená vo Výpise 1. Táto registrácia je vykonávaná v triede *Startup*, prostredníctvom metódy *ConfigureContainer*.

```
public void ConfigureContainer(ContainerBuilder builder)
{
    builder.RegisterType<LoggerService>().As<ILoggerService>();
    builder.RegisterType<DapperConnection>().As<IDapperConnection>();
}
```

Výpis 1: Autofac IoC registrácia

Všetky závislosti v aplikácií sú injektované prostredníctvom konštruktora. Pokiaľ je komponenta zaregistrovaná v IoC kontajneri, môže byť následne využitý prístup z Výpisu 2.

```
private readonly ILoggerService _logger;

public InterceptorService(ILoggerService logger) {
    _logger = logger;
}
```

Výpis 2: Autofac IoC constructor injection

AutoMapper

Ako už z názvu vyplýva, jedná sa o knižnicu, ktorá slúži na mapovanie jedného typu objektu na iný, bez nutnosti písania rozsiahleho a opakujúceho sa kódu. AutoMapper je prístupný prostredníctvom IoC kontajnera a umožňuje vytvárať vlastné konfigurácie mapovania, viď Výpis 3. Všetky konfigurácie sú uložené v triede *AutoMapperProfile* a zaregistrované v triede *Startup*. [10]

```
CreateMap<EndPoint, EndPointDTO>().ReverseMap();
CreateMap<Robot, RobotDTO>()
    .ForMember(dest => dest.Password, opt => opt.Ignore())
    .ForMember(dest => dest.UserName, opt => opt.Ignore());
```

Výpis 3: AutoMapper konfigurácia

V aplikácií je táto knižnica využívaná na mapovanie medzi entitami a data transfer objektami. Ukážku použitia zobrazuje Výpis 4.

```
List<RobotDTO> robotDtos = _mapper.Map<List<RobotDTO>>(robots);
```

Výpis 4: AutoMapper použitie

DbUp

DbUp predstavuje balíček, ktorý slúži na zavádzanie zmien do SQL databáz. Sleduje, ktoré skripty boli už vykonané a aktualizuje databázu pomocou nových. Všetky skripty musia byť uložené v priečinku s názvom *Scripts*. Jednotlivé skripty musia mať jedinečný názov a typ buildu nastavený na *Embedded Resource*. [11]

Pri každom spustení aplikácie dochádza ku kontrole stavu databázy a prípadnej aktualizácií. Túto činnosť vykonáva trieda *DbUpdater* prostredníctvom metódy *Update*, ktorá je zobrazená vo Výpise 5.

```
public void Update()
{
    var connectionString = _configuration["ConnectionString:WSConnection"];

    var upgrader = DeployChanges.To.PostgresqlDatabase(connectionString)
                .WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly())
                .WithVariablesDisabled().WithTransactionPerScript()
                .LogToConsole().Build();

    upgrader.PerformUpgrade();
}
```

Výpis 5: DbUp aktualizácia

Dapper

Jedná sa o knižnicu pre objektovo relačné mapovanie, t.j. poskytuje mapovanie medzi objektami a databázou. Dapper sa vyznačuje jednoduchým použitím a vysokým výkonom. Je označovaný aj ako kráľ mikro ORM pre jeho rýchlosť. Poskytuje metódy pre CRUD operácie bez nutnosti písania databázových skriptov. Ďalej obsahuje metódy ako *Query* alebo *Execute*, ktoré umožňujú zavolať uložené procedúry, poprípade vykonať parametrizované SQL príkazy. [12]

Tieto a ďalšie poskytované metódy sú v aplikácií obalené prostredníctvom generického repozitára s názvom *BaseRepository*.

SignalR

Jedná sa o knižnicu, ktorá umožňuje vytvárať real-time webové aplikácie. Princíp spočíva vo vytvorení komunikačného kanálu medzi serverom a klientmi a následnom odosielaní dát zo servera smerom ku jednotlivým klientom. SignalR podporuje nasledujúce typy komunikácie: *WebSocket*, *Server-Sent Events*, *Long Pooling* a dokáže vybrať najvhodnejší spôsob na základe možností servera a klienta. [13]

Základný prvok predstavuje *Hub*, ktorý umožňuje serveru volať metódy klienta a naopak. Dáta sú odosielané v binárnej alebo textovej podobe. Jednotlivé Huby je potrebné pridať do *routingu* aplikácie v triede *Startup*. Ukážka tejto konfigurácie je zobrazená vo Výpise 6.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<MobileDeviceHub>("/hub/mobileDeviceHub");
    endpoints.MapHub<CustomDataHub>("/hub/customDataHub");
    endpoints.MapHub<DashboardHub>("/hub/dashboardHub");
});
```

Výpis 6: SignalR - routing registrácia

Všetky Huby dedia z abstraktnej triedy *BaseHub*, ktorá obsahuje metódy pre pripojenie a odpojenie klienta a taktiež autorizačný atribút. Ukážka Hubu je zobrazená vo Výpise 7. Táto trieda neobsahuje žiadnu implementáciu, pretože klienti nemôžu prevolávať metódy servera.

```
public class CustomDataHub : BaseHub<ICustomDataHub>
{
    public CustomDataHub(ISignalRConnectionManager signalRConnectionManager) :
        base(signalRConnectionManager){}
}
```

Výpis 7: SignalR - ukážka triedy CustomDataHub

Aby bolo možné odosielať dáta zo servera smerom ku jednotlivým klientom, musí sa do Hubu zaregistrovať rozhranie s požadovanými metódami, viď Výpis 8.

```
public interface ICustomDataHub
{
    Task ReceiveAlarms(IEnumerable<RobotAlarmDTO> alarms);
    Task ReceiveCustomData(IEnumerable<RobotDataDTO> robotDatas);
    Task ReceiveStatuses(RobotCommunicationHistoryDTO statuses);
}
```

Výpis 8: SignalR - ukážka rozhrania ICustomDataHub

Následne klienti, ktorí chcú získavať dáta z tohoto Hubu, musia implementovať metódy s rovnakou signatúrou. Ukážka je zobrazená vo Výpise 15.

Tmds.MDns

Táto knižnica slúži na vyhľadávanie služieb v lokálnej sieti pomocou mDNS protokolu. Obsahuje udalosti pridania, zmeny alebo odstránenia vyhľadaných služieb. Ďalej automaticky vracia IP adresu a názov hostiteľa. [14]

Aplikácia využíva túto knižnicu pre automatické vyhľadávanie robotov. Detailnejší popis je uvedený v Sekcii 3.1.3.

Html Agility Pack

Jedná sa o knižnicu, ktorá umožňuje parsrovať HTML dokumenty. Dovoľuje pristupovať k jednotlivým elementom a atribútom v dokumente, poprípade získať dáta prostredníctvom jazyka *XPath*[10]. [15]

Aplikácia využíva tento balíček pre parsrovanie odpovedí prichádzajúcich z jednotlivých robotov, viď Sekcia 3.1.4.

3.1.2 Logovanie

Logovanie je neoddeliteľnou súčasťou každého softvéru. Aplikácia využíva princíp aspektovo orientovaného programovania pre zachytávanie neočakávaných výnimiek, ktoré nastanú počas behu. Princíp AOP spočíva v úprave chovania existujúceho kódu bez nutnosti priameho zásahu. Informácie o neočakávaných udalostiach sú ukladané do databázy.

Základné prvky tvoria triedy *InterceptorService* a *LoggerService*. Trieda InterceptorService obaľuje volania metód pomocou *try - catch* bloku, odchytáva prípadné výnimky a vykonáva logovanie prostredníctvom triedy LoggerService, viď Výpis 9. Trieda LoggerService obsahuje metódy *Debug, Info* a *Error* pre rôzne úrovne logovania.

[10]XPath predstavuje jazyk, ktorý umožňuje vyberať konkrétne časti XML dokumentu.

```
public void Intercept(IInvocation invocation)
{
    try
    {
        invocation.Proceed();
    }
    catch (Exception ex)
    {
        var args = invocation.Arguments.IsNullOrEmpty() ? null : JsonConvert.
            SerializeObject(invocation.Arguments);

        var logData = new LogModel
        {
            Args = args ?? string.Empty,
            Method = invocation.Method?.Name ?? string.Empty,
            ClassType = invocation.TargetType?.Name ?? string.Empty,
            Message = ex.Message ?? string.Empty,
            StackTrace = ex.StackTrace ?? string.Empty,
            ExceptionMessage = ex.InnerException?.Message ?? string.Empty
        };

        _logger.Error(logData);
    }
}
```

Výpis 9: Logovanie - ukážka metódy Intercept

Každá komponenta, ktorá chce využívať tento spôsob logovania, musí byť zaregistrovaná v IoC kontajneri. Túto funkcionalitu zabezpečuje rozšírenie knižnice Autofac prostredníctvom metód *EnableInterfaceInterceptors* a *InterceptedBy*. Ukážka registrácie je zobrazená vo Výpise 10.

```
builder.Register(x => new InterceptorService(x.Resolve<ILoggerService>()));
builder.RegisterType<RobotProvider>().EnableInterfaceInterceptors().
    InterceptedBy(typeof(InterceptorService)).As<IRobotProvider>();
```

Výpis 10: Logovanie - Autofac IoC registrácia

3.1.3 Vyhľadávanie robotov

Automatické vyhľadávanie robotov malo predstavovať jednu z podporných funkcionalít servera pri pridávaní robotov. Počas vývoja som však narazil na problém pri detekcii robotov v lokálnej sieti prostredníctvom mDNS protokolu. K vyhľadaniu robota dochádzalo v nepravidelných a náhodných intervaloch. Mnohokrát nebolo možné robota vôbec detekovať a objaviteľný bol až po následnom reštarte. Testovanie bolo vykonané pomocou viacerých nástrojov a knižníc, aby sa vylúčila chyba na strane softvéru. Taktiež boli testované rozdielne verzie RobotWare, robot v stave nečinnosti, ako aj počas behu. Na Obrázku 23 môžete vidieť výsledok vyhľadávania. Dochádzalo k opakovanému pridávaniu a odoberaniu robota až sa vyhľadávanie úplne zastavilo a bol potrebný reštart pre jeho opätovné nájdenie.

```
Timestamp     A/R    Flags  if Domain               Service Type         Instance Name
11:28:55.566  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:29:41.855  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:30:56.270  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:31:37.966  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:32:12.895  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:32:55.392  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:35:00.588  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:35:42.152  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:37:32.288  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:38:14.688  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:39:03.925  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:40:21.427  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:41:06.650  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
11:41:49.977  Rmv        0   6 local.              _http._tcp.          RobotWebServices_100950
11:43:08.338  Add        2   6 local.              _http._tcp.          RobotWebServices_100950
```

Obr. 23: Vyhľadávanie robotov v sieti pomocou mDNS protokolu

Keďže nebolo možné získať zoznam pripojených robotov na vyžiadanie, je vyhľadávanie spustené po štarte servera na pozadí. Po nájdení robota dochádza k pridaniu IP adresy a názvu do slovníka. Túto funkcionalitu zabezpečuje trieda *RobotSearchService* prostredníctvom metódy *_OnServiceAdded*, viď Výpis 11.

```
private void _OnServiceAdded(object sender, ServiceAnnouncementEventArgs e)
{
    var service = e.Announcement;
    if (service.IsNull() || !service.Instance.Contains(StringConstants.
        RobotAutoSearchServiceKey) return;

    var host = service.Hostname;
    var ipAddresses = service.Addresses;
    var port = service.Port;


    foreach(var ipAddress in ipAddresses)
```

```
    {
        var autosearchModel = new RobotAutosearchModel(ipAddress.ToString(),
            port, host);
        _robotDic.AddOrUpdate($"{ipAddress}:{port}", autosearchModel, (oldKey,
            oldValue) => autosearchModel);
    }
}
```

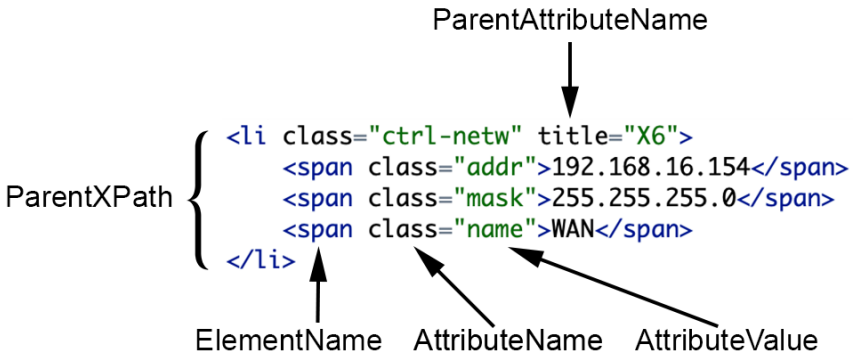
Výpis 11: Vyhľadávanie robotov

3.1.4 Parsrovanie dát

Dáta získané z robotov je potrebné previesť do takej podoby, aby mohli byť uložené do databázy a preposlané jednotlivým klientom. Pre tento účel je možné priradiť každému koncovému bodu parsre, ktoré dokážu vybrať potrebné údaje z XHTML odpovede. Túto činnosť vykonáva extension metóda *ToParsedValues* v triede *HtmlDocumentExtensions*. Parser má nasledujúcu podobu:

- ElementName
- AttributeName
- AttributeValue
- ParentXPath
- ParentAttributeName

Na Obrázku 24 môžete vidieť ukážku odpovede robota a jej napárovanie na jednotlivé časti parseru.



Obr. 24: Napárovanie odpovede na parser

V Tabuľke 3 môžete vidieť parsre pre získanie dát z odpovede uvedenej na Obrázku 24.

Tabuľka 3: Parsre pre získanie dát z odpovede

ElementName	AttributeName	AttributeValue	ParentAttri-buteName	ParentXPath
span	class	addr	title	//li[@class="ctrl-netw"]
span	class	mask	title	//li[@class="ctrl-netw"]
span	class	name	title	//li[@class="ctrl-netw"]

Rozparsrovaná odpoveď je reprezentovaná vo formáte *kľúč - hodnota*. Každý záznam ďalej obsahuje názov rodiča, aktuálny čas, referenciu na koncový bod a robota. Tabuľka 4 obsahuje výslednú podobu dát po rozparsrovaní.

Tabuľka 4: Rozparsrovaná odpoveď

Názov	Kľúč	Hodnota
X6	addr	192.168.16.154
X6	mask	255.255.255.0
X6	name	WAN

3.1.5 RobotService

Jedná sa o hlavnú službu, ktorá je spúšťaná pri štarte aplikácie a zabezpečuje nasledujúcu funkcionalitu:

- Získanie zoznamu robotov a naštartovanie komunikácie.
- Pridanie, odobratie alebo aktualizovanie komunikácie s robotmi.
- Uloženie získaných dát a ich predanie triede *SignalRService*.
- Uloženie zmien stavu komunikácie a ich predanie triede *SignalRService*.
- Obnovenie komunikácie pri výpadku spojenia.

Vytvoriť komunikáciu s robotom má za úlohu metóda *_AddRobotCommunications*. Dochádza tu k načítaniu koncových bodov robota, vytvoreniu HTTP a WebSocket klienta pre daný typ autentifikácie a v neposlednom rade k samotnému naštartovaniu komunikácie. Ďalej je tu vykonávaná registrácia udalostí pre získanie dát a zmien stavov pripojenia. Ukážka je zobrazená vo Výpise 12.

```
private void _AddRobotCommunications(params Robot[] robots)
{
    foreach (var robot in robots)
    {
        if (robot.IsNull() || robot.ConnectionType == ConnectionTypeEnum.None)
            continue;

        var endPoints = _endPointRepository.GetWithParsersAndParams(robot.Id);
        if (endPoints.IsNullOrEmpty()) continue;

        var lastAlarmNumbers = _robotAlarmRepository.GetLastAlarms(robot.Id).
            ToConcurrentDictionary();

        var communicationModel = new CommunicationModel(robot, endPoints,
            lastAlarmNumbers);
        var authModel = new RobotAuthModel(robot.UserName, DataProtectionHelper
            .Decrypt(robot.Password));

        var httpOperations = _newhttpOperationsFactories[robot.AuthType].Create
            (authModel);
        var wsHttpClient = _newWsHttpClient(httpOperations);
        var wsSocket = _newWsSocket(wsHttpClient, communicationModel);

        var robotCommunication = _newRobotCommunication(communicationModel,
            wsHttpClient, wsSocket);
        _repeatCommunicationDictionary.TryRemove(robot.Id, out _);

        _robotCommunications.Add(robotCommunication);

        robotCommunication.OnRobotDataReceived += OnRobotDataReceived;
        robotCommunication.OnRobotCommunicationChange +=
            OnRobotCommunicationChange;
        robotCommunication.Start();
    }
}
```

Výpis 12: Vytvorenie komunikácie s robotom

3.1.6 RobotCommunication

Táto trieda slúži na priamu komunikáciu s robotmi prostredníctvom HTTP požiadaviek a WebSocket protokolu. Medzi jej hlavné úlohy patria:

- Štart, aktualizácia alebo zastavenie komunikácie s robotom.
- Inicializácia časovačov pre rôzne priority koncových bodov.
- Pravidelné pridávanie koncových bodov do fronty pre HTTP požiadavky.
- Získavanie stavu komunikácie a jej následné predanie triede *RobotService*.
- Rozparsrovanie získaných dát a ich následné predanie triede *RobotService*.

V tejto triede dochádza k súčasnému behu viacerých vlákien a preto bolo potrebné vyriešiť bezpečné zdieľanie prostriedkov. Jedná sa konkrétne o frontu koncových bodov, kde dochádza k jej pravidelnej aktualizácií prostredníctvom časovačov a súčasne časovač pre HTTP požiadavky odoberá prvky z tejto fronty. Pre takéto prípady poskytuje .NET framework *thread-safe* kolekcie, medzi ktorými sa nachádza aj implementácia fronty s názvom *ConcurrentQueue*[11].

Jednu z hlavných metód predstavuje *OnHttpRobotDataReceived*, ktorá ma za úlohu vybrať koncový bod z fronty, poslať HTTP požiadavku a rozparsrovať získanú odpoveď. Keďže pri častom dotazovaní hrozí pozastavenie získavania dát z robota, je ďalšia požiadavka odoslaná až po spracovaní predchádzajúcej. Ukážka tejto metódy je zobrazená vo Výpise 13.

```
private async void OnHttpRobotDataReceived(object sender, ElapsedEventArgs args
    ) {
    var isServiceUnavailable = false;
    _getRequestTimer.Stop();
    if (!_requestQueue.TryDequeue(out EndPoint endpoint)) return;

    try
    {
        var request = _GetHttpRequestModel(endpoint);
        var response = await _httpClient.GetResponseAsync(request);

        if (response.StatusCode == HttpStatusCode.Unauthorized)
        {
            _InvokeCommunicationChangeEvent(RobotCommunicationStatusEnum.
                Unauthorized);
            return;
        }
```

[11] https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe

```
        if (response.StatusCode == HttpStatusCode.ServiceUnavailable)
        {
            _InvokeCommunicationChangeEvent(RobotCommunicationStatusEnum.
                HttpUnavailable, endpoint);
            _getRequestTimer.Pause(TimeSpan.FromMinutes(2).TotalMilliseconds);
            _requestQueue.Enqueue(endpoint);
            isServiceUnavailable = true;
            return;
        }

        response.EnsureSuccessStatusCode();
        _ParseSuccessResponse(response, endpoint);
    }
    catch (Exception ex)
    {
        _InvokeCommunicationChangeEvent(RobotCommunicationStatusEnum.
            HttpRequestError, endpoint, ex.Message);
    }
    finally
    {
        if (!isServiceUnavailable && !_requestQueue.IsEmpty)
            _getRequestTimer.Start();
    }
}
```

Výpis 13: Získanie dát z robota

3.1.7 SignalRService

Trieda SignalRService zabezpečuje komunikáciu s klientmi prostredníctvom SignalR protokolu. Obsahuje metódy *SendCommunicationStatusChange*, *SendRobotAlarm* a *SendRobotData* pre odosielanie rozličných typov dát. Pre získanie zoznamu pripojených klientov je využívaná trieda *SignalRConnectionManager*. Ukážka odoslania alarmov jednotlivým klientom je zobrazená vo Výpise 14.

```
public async Task SendRobotAlarm(List<RobotAlarm> robotAlarms, Robot robot)
{
    var excludedConnections = _GetExcludedConnections(robot.Id);
    if (excludedConnections.IsNull()) return;
```

```
    var alarmsDto = _mapper.Map<List<RobotAlarmDTO>>(robotAlarms);
    if (alarmsDto.IsNullOrEmpty()) return;

    await _cutomDataHub.Clients.AllExcept(excludedConnections).ReceiveAlarms(
        alarmsDto);
    await _mobileDeviceHub.Clients.AllExcept(excludedConnections).ReceiveAlarms
        (alarmsDto);
}
```

Výpis 14: SignalRService - odoslanie alarmov

Ďalej táto trieda zabezpečuje, aby boli dáta odoslané iba klientom, ktorí majú väzbu na robota. Zoznam klientov, ktorým nemajú byť odoslané dáta je získavaný pomocou metódy *_GetExcludedConnections*.

3.2 HTML stránka

Pri HTML stránke došlo k jej presunutiu z robota na stranu servera do priečinku *wwwroot*. Jedná sa o špeciálnu zložku, ktorá umožňuje pristupovať k statickým súborom servera prostredníctvom HTTP požiadaviek. Táto stránka je dostupná pod adresou *"adresa serveru"/ws/index.html*. Pre načítanie dát je navyše potrebné poslať do danej url adresy nasledujúce parametre:

- robotId
- userId

Ďalej došlo k menším úpravám na strane kódu. Jedná sa prevažne o úpravu HTTP klienta, ktorý teraz komunikuje s novým API pre získavanie potrebných dát. Taktiež bolo potrebné upraviť funkcie, ktoré vykresľujú získavané informácie, pretože došlo k zmene dátových modelov.

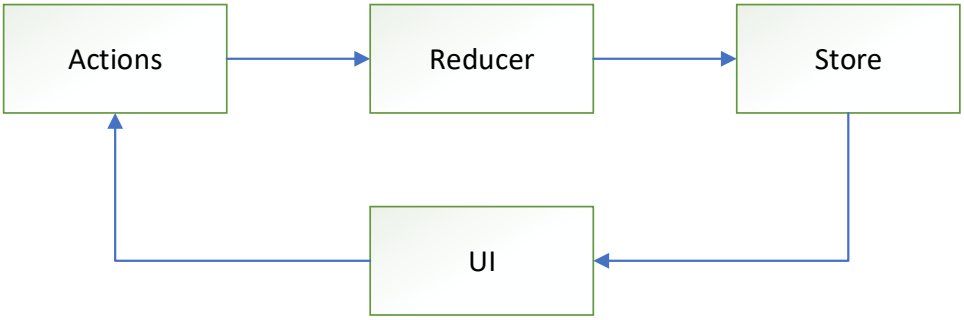
3.3 Klient

3.3.1 Použité knižnice

Pridávanie knižníc do *React* aplikácií je vykonávané prostredníctvom *Node Package Managera*. Jedná sa o správcu JavaScriptových balíčkov, ktorý umožňuje ich jednoduchú inštaláciu a údržbu. Novú knižnicu je možné pridať pomocou príkazu *npm install "názov knižnice"*. [16]

Redux

Jedná sa o stavový kontajner pre JavaScript aplikácie. Táto knižnica uľahčuje spravovať zobrazené informácie a reagovať na akcie užívateľov. Princíp fungovania je zobrazený na Obrázku 25.



Obr. 25: Redux - princíp fungovania

Základ knižnice tvorí *Store*, ktorý uchováva stav celej aplikácie a je prístupný všetkým komponentám. Tento prístup zabezpečuje, že nie je potrebné neustále prenášať stav z jednej komponenty do druhej.

Ďalej sa tu nachádzajú *Actions*, pomocou ktorých aplikácia dokáže odosielať údaje do *Storu*. Jedná sa o jediný spôsob, akým je možné vykonávať zmeny stavu aplikácie. Jednotlivé akcie sú volané prostredníctvom špeciálnej metódy s názvom *dispatch*. Každá akcia predstavuje jednoduchý objekt, ktorý obsahuje typ a samotný výpočet.

Posledný prvok predstavuje *Reducer*. Jedná sa o funkcie, ktoré prijímajú predchádzajúci stav aplikácie a vrátia nový stav na základe akcie, ktorá im bola odovzdaná. Jedná sa o čisté funkcie, ktoré pre danú množinu argumentov vrátia vždy rovnaký výsledok. [17]

Antd

Táto knižnica poskytuje komponenty užívateľského rozhrania. Nachádza sa tu široká škála prvkov, od jednoduchých tlačidiel, cez rôzne formuláre až po komplexné tabuľky. Výhodou je taktiež TypeScript podpora a vysoká úroveň prispôsobenia. [18]

Axios

Jedná sa o HTTP klienta pre *Node.js* aplikácie. Obsahuje implementáciu nielen pre základné typy požiadaviek (put, get, post, delete) ale podporuje aj súbežné dotazovanie a rozsiahlu konfiguráciu. Samozrejmosťou je aj odchytávanie výnimiek a podpora pre TypeScript. [19]

V aplikácií je táto knižnica rozšírená o automatickú obnovu tokenu. Túto činnosť vykonáva trieda *httpClientHelper*.

Sass

Sass rozširuje jazyk CSS o premenné, funkcie alebo dedičnosť. Táto knižnica pomáha organizovať a zdieľať štýly v rámci aplikácie. [20]

Každá stránka klienta obsahuje vlastnú sadu štýlov, ktoré sú uložené v súbore s nasledujúcou podobou: *"názov stránky".module.scss*.

@microsoft/signalr

Táto knižnica slúži na nadviazanie spojenia so serverom prostredníctvom SignalR protokolu. [21]

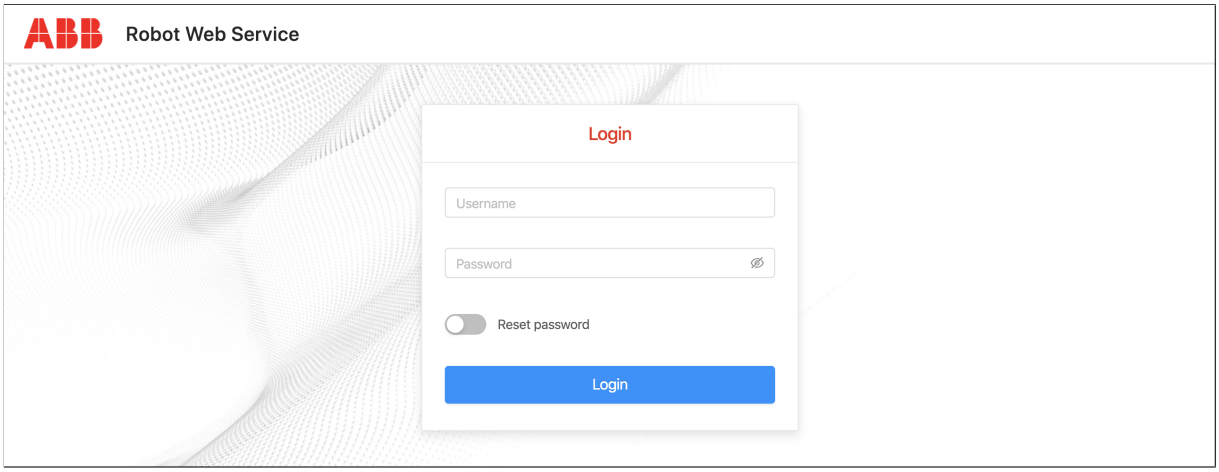
Ukážka pripojenia je zobrazená vo Výpise 15. Môžete tu vidieť, že klient čaká na nové alarmy prichádzajúce z *CustomDataHubu* prostredníctvom metódy *ReceiveAlarms*.

```
let connection = getSignalRClient(urlData.customDataHub)

connection.on("ReceiveAlarms", (data: AlarmsStore.Alarm[]) => {
    if(this.props.isAlarmLoading || this.state.isFiltered) return

    this.props.signalRUpdate(data, this.props.alarms)
})

connection.start().catch(function (err) {
    error(notificationData.alarms.signalrerror.title, notificationData.alarms.
        signalrerror.message)
});
```

Výpis 15: SignalR - ukážka pripojenia klienta

3.3.2 Prihlásenie

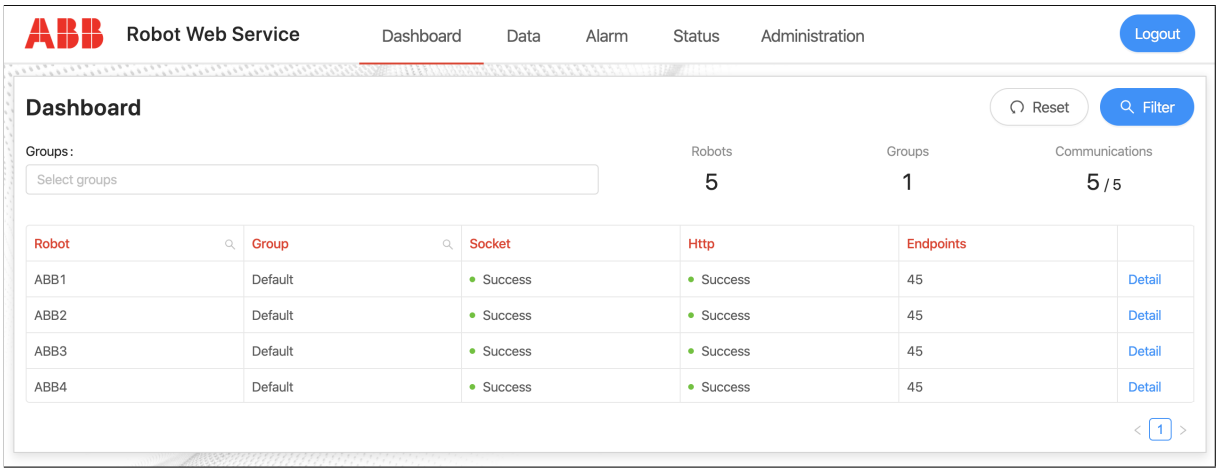
Pre vstup do systému je potrebné vykonať prihlásenie. Ukážka prihlasovacej obrazovky je zobrazená na Obrázku 26. Aplikácia obsahuje prednastavený administrátorský účet s názvom *admin*. Pri prvom prihlásení je potrebné nastaviť heslo pre tento účet prostredníctvom sekcie obnovenia. Minimálna dĺžka hesla je osem znakov, pričom musí ďalej obsahovať aspoň jednu číslicu, jedno veľké písmeno, jedno malé písmeno a jeden špeciálny znak.



Obr. 26: Klient - prihlásenie

3.3.3 Dashboard

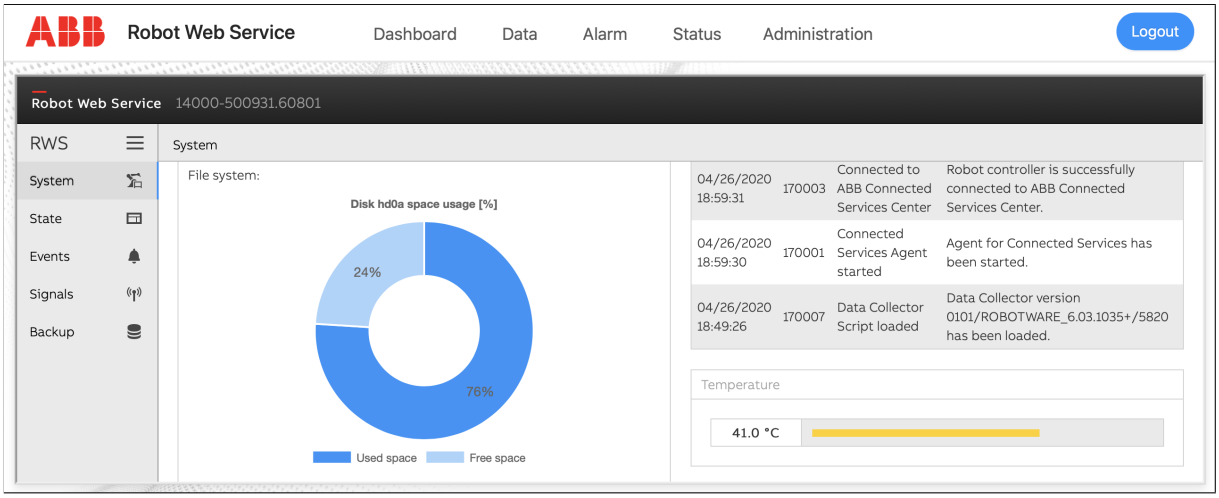
Jedná sa o domovskú stránku aplikácie. Nachádza sa tu prehľad robotov priradených k užívateľovi, viď Obrázok 27. Stránka zobrazuje celkový počet skupín, robotov a aktívnych komunikácií. Každý robot obsahuje informáciu o stave HTTP a WebSocket pripojenia, ktorá je aktualizovaná prostredníctvom SignalR protokolu. Stavy môžu nadobúdať nasledujúce hodnoty: *Disabled, Waiting, Error* a *Success*. Taktiež je možné vykonávať filtrovanie na základe skupín.



Obr. 27: Klient - dashboard

3.3.4 Detail

Pre zobrazenie detailu robota slúži existujúca HTML stránka. Tá je vykresľovaná prostredníctvom elementu s názvom *iframe*[12]. Ukážka je zobrazená na Obrázku 28.

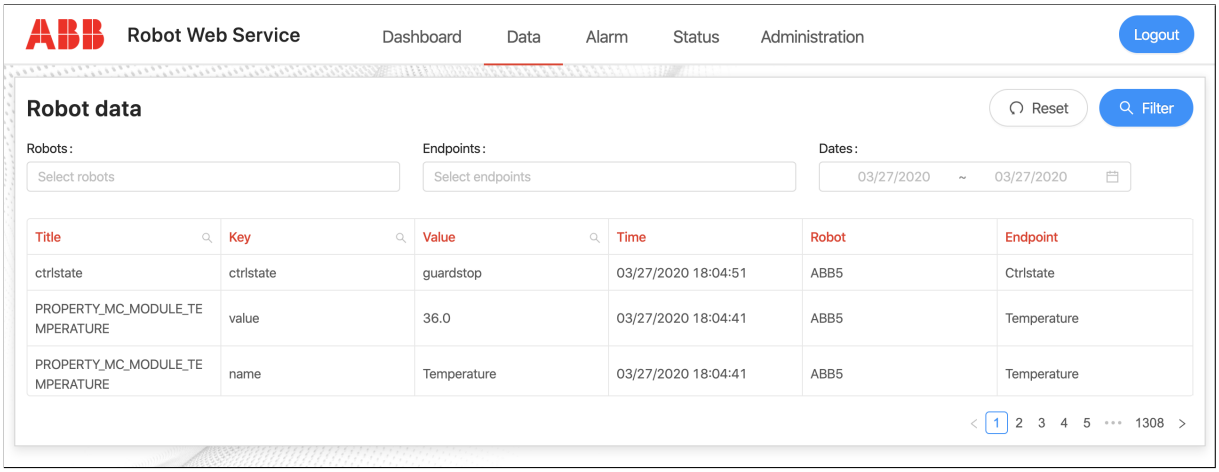


Obr. 28: Klient - detail robota

[12]Iframe umožňuje zobrazenie webovej stránky v rámci webovej stránky.

3.3.5 Dáta

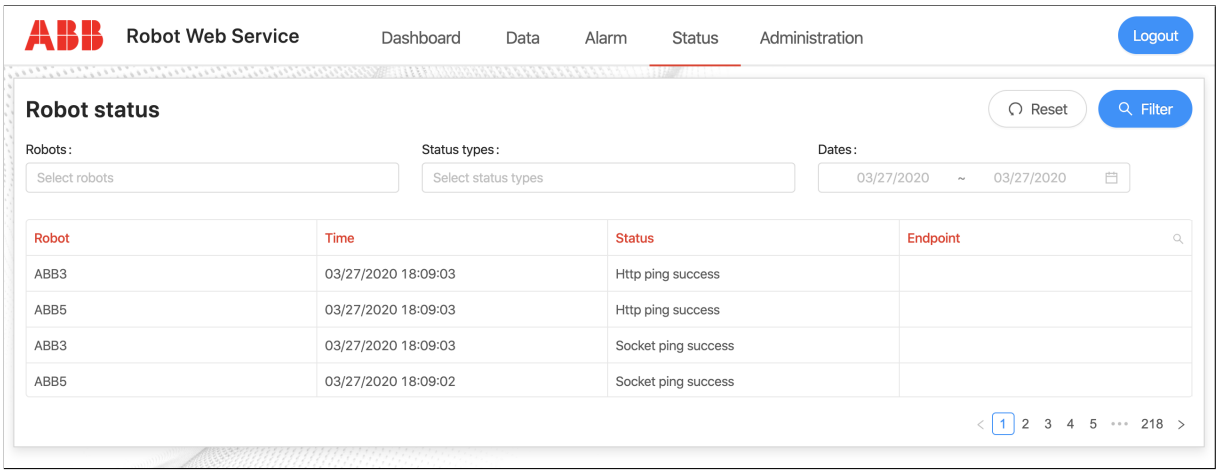
Táto stránka zobrazuje dáta pochádzajúce z jednotlivých koncových bodov. Dáta je možné filtrovať na základe robotov, kategórií a dátumu. Aktualizácia tabuľky prebieha pomocou SignalR protokolu. Ukážku stránky môžete vidieť na Obrázku 29.



Obr. 29: Klient - dáta

3.3.6 Status

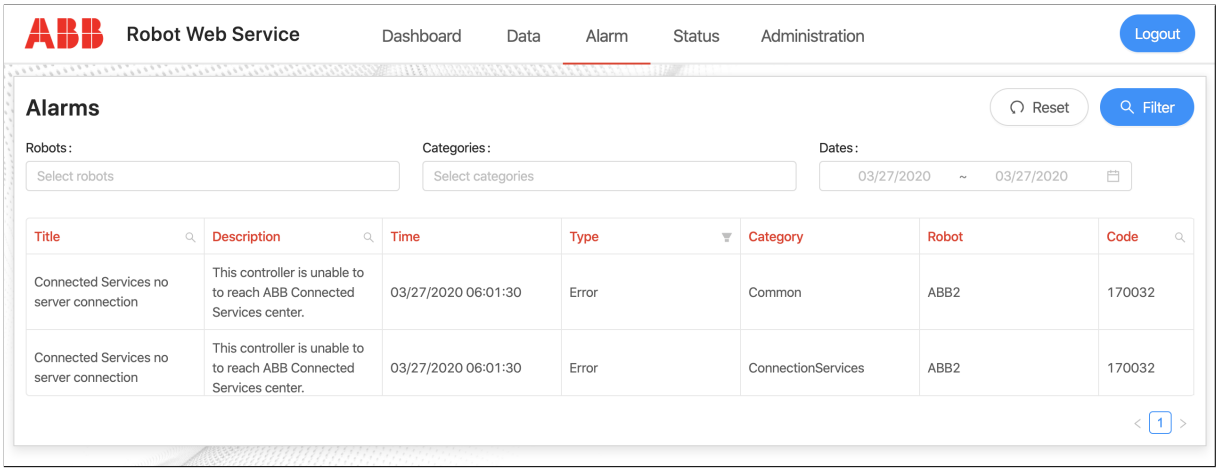
Užívateľ má taktiež k dispozícii stránku, ktorá ponúka prehľad priebehu komunikácie s robotmi. Samozrejmosťou je filtrovanie podľa robotov, statusov a dátumu. Nové záznamy sú načítavané s využitím SignalR protokolu. Ukážka je zobrazená na Obrázku 30.



Obr. 30: Klient - status

3.3.7 Alarmy

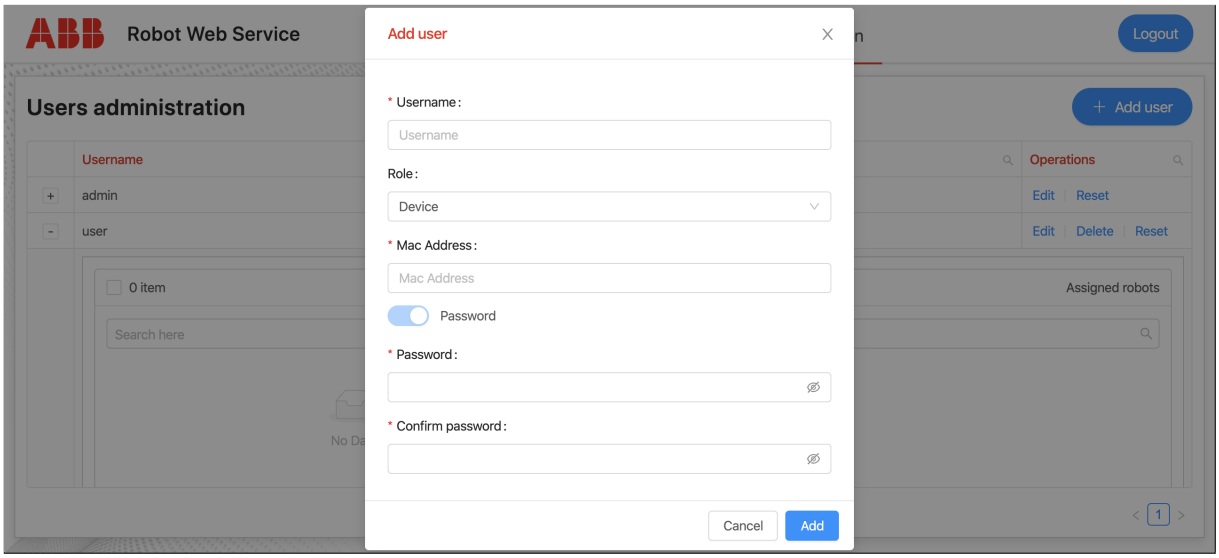
Na Obrázku 31 môžete vidieť alarmy z jednotlivých robotov. Aktualizácia prebieha prostredníctvom SignalR protokolu. Ďalej sa tu nachádza filtrovanie podľa robotov, kategórií a dátumu.



Obr. 31: Klient - alarmy

3.3.8 Administrácia užívateľov

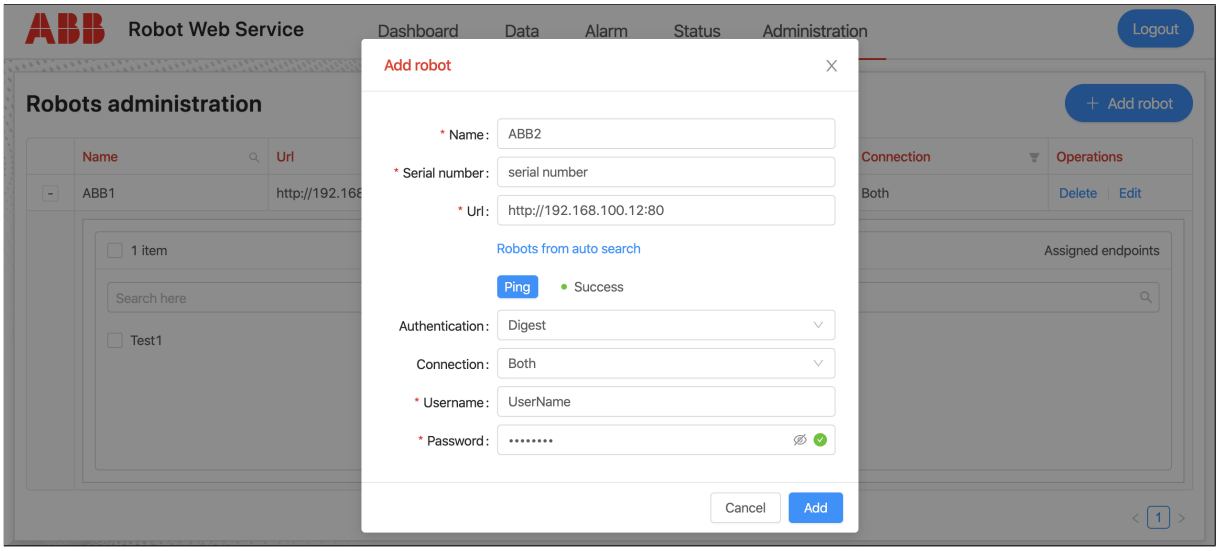
Táto stránka umožňuje vytvárať, editovať alebo mazať užívateľov. Ďalej je možné priraďovať robotov k jednotlivým užívateľom a vykonávať reset hesla. K nastaveniu hesla dochádza pri prvom prihlásení alebo po resetovaní administrátorom. Výnimku tvoria užívatelia mobilných zariadení, kde všetky prihlasovacie údaje nastavuje administrátor. Na Obrázku 32 je zobrazený formulár pre vytvorenie nového užívateľa.



Obr. 32: Klient - administrácia užívateľov

3.3.9 Administrácia robotov

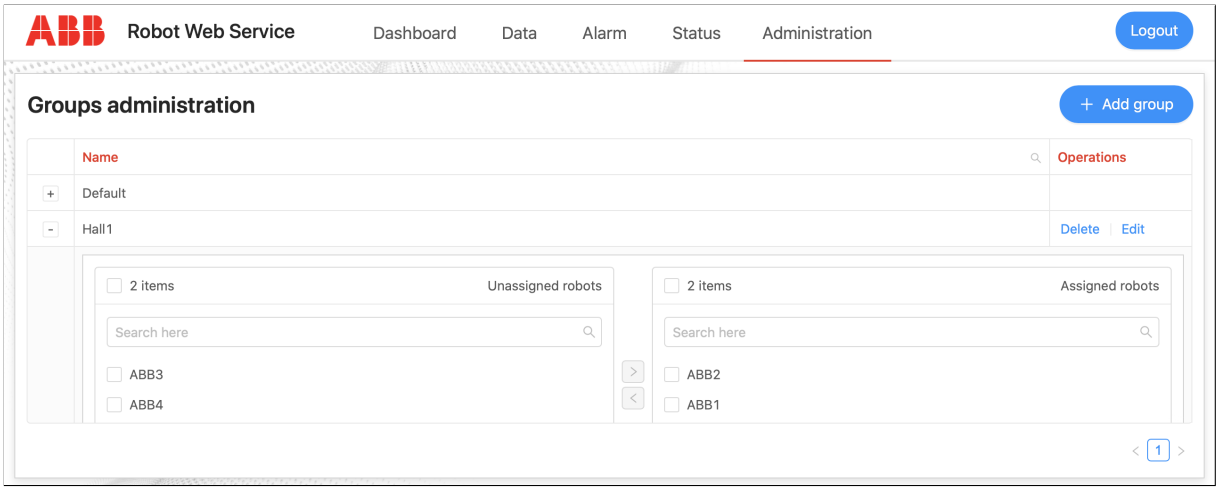
Prostredníctvom tejto stránky dochádza k vytváraniu, aktualizácií alebo mazaniu robotov. Na Obrázku 33 je zobrazený formulár pre pridanie robota. Tento formulár poskytuje zoznam vyhľadaných robotov v sieti a taktiež umožňuje otestovať dostupnosť pridávaného robota. Poslednou funkcionalitou je priraďovanie koncových bodov jednotlivým robotom.



Obr. 33: Klient - administrácia robotov

3.3.10 Administrácia skupín

Stránku pre administráciu skupín môžete vidieť na Obrázku 34. Aplikácia obsahuje preddefinovanú skupinu s názvom *Default*. Do tejto skupiny sú automaticky priradení noví roboti, poprípade roboti bez zaradenia. Jeden robot môže byť súčasťou viacerých skupín.



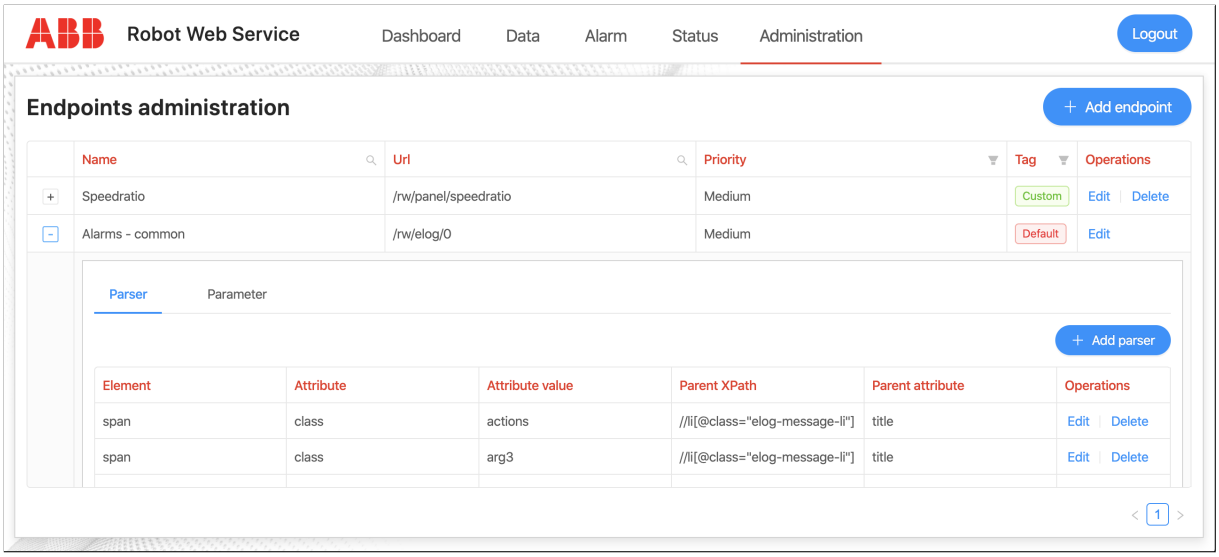
Obr. 34: Klient - administrácia skupín

3.3.11 Administrácia koncových bodov

V tejto časti dochádza k pridávaniu, aktualizácií alebo odstraňovaniu koncových bodov. Aplikácia obsahuje dva typy koncových bodov:

- **Default:** Predvolené koncové body, ktoré sú vyžadované pre plnú funkcionalitu aplikácie. Po pridaní nového robota dochádza k ich automatickému priradeniu k tomuto robotovi.
- **Custom:** Jedná sa o koncové body vytvorené administrátorom. Údaje získané prostredníctvom týchto koncových bodov sú užívateľovi k dispozícií v sekcii Dáta.

Ďalej je možné vytvárať, editovať alebo mazať parsre a parametre pre jednotlivé koncové body. Ukážka je zobrazená na Obrázku 35.



Obr. 35: Klient - administrácia koncových bodov

3.3.12 Ostatné

Súčasťou klienta sú globálne notifikácie, ktoré užívateľa informujú o úspešných operáciach a prípadných chybách. Všetky formuláre obsahujú validácie pre prázdne vstupy a v prípade hesiel aj kontrolu zhody. Vybrané stĺpce tabuliek umožňujú vyhľadávanie na základe textu. Pri dlhotrvajúcich operáciach dochádza k zobrazeniu indikátora, ktorý informuje užívateľa, že sa niečo deje.

3.4 iOS a tvOS aplikácia

Pri týchto aplikáciach došlo k viacerým zmenám, ktoré súvisia s novým API pre získavanie dát z robotov. Jedná sa o nasledujúce úpravy:

- Autentifikácia prostredníctvom JWT tokenu.

- Úprava dátových modelov a vrstvy pre webovú komunikáciu.
- Pridanie SignalR komunikácie.
- Úprava užívateľského rozhrania.
- Odstránenie manuálneho pridávania robotov a skupín.
- Zmena vzťahu skupiny a robota z 1:N na M:N a s tým súvisiace úpravy databázovej vrstvy.
- Pridanie synchronizácie skupín a robotov s webovým serverom.

3.4.1 Použité knižnice

Knižnice do Swift projektov je možné pridávať prostredníctvom správcu závislostí *CocoaPods*. Všetky závislosti sa nachádzajú v špeciálnom súbore s názvom *Podfile*. Inštalácia následne prebieha pomocou príkazu *pod install*. [22]

Alamofire
Jedná sa o knižnicu pre sieťovú komunikáciu s využitím HTTP požiadaviek. [23]

V aplikácií je táto knižnica obalená generickými metódami *request, getItems* a *getSingleItem* v triede *HttpClient*, ktorá nahradila predchádzajúceho sieťového klienta. Ďalšiu dôležitú triedu predstavuje *TokenAuthentificationHandler*, ktorá sa stará o autentifikáciu každej požiadavky a prípadnú obnovu tokenu.

KeychainSwift
Táto knižnica poskytuje pomocné metódy pre prácu s *KeyChain* úložiskom. KeyChain predstavuje bezpečné miesto pre uloženie hesiel, kreditných kariet a iných citlivých informácií. Po uložení sú tieto údaje k dispozícií iba pre danú aplikáciu. [24]

Táto knižnica je v aplikácií využívaná pre ukladanie autentifikačných tokenov a prihlasovacích údajov servera.

SwiftyJSON
SwiftyJSON predstavuje knižnicu pre mapovanie JSON objektov na vlastné modely bez nutnosti písania nadbytočného kódu. Podporuje vytváranie vlastných konfigurácií mapovania, kolekcie alebo vnorené objekty. [25]

V aplikácii je táto knižnica využívaná triedou *HttpClient* pre mapovanie JSON odpovede na očakávaný model.

SwiftSignalRClient

Jedná sa o knižnicu, ktorá umožňuje nadviazať komunikáciu so serverom prostredníctvom SignalR protokolu. [26]

O túto funkcionalitu sa stará trieda *SignalRService*, ktorá obsahuje metódy pre začatie a ukončenie komunikácie. Pomocou SignalR pripojenia je možné získavať nové alarmy, signály alebo stav komunikácie pre jednotlivých robotov.

RealmSwift

Jedná sa o objektovo orientovanú databázu pre lokálne ukladanie dát. Poskytuje širokú škálu operácií pre manipuláciu s týmito údajmi. Vytvorenie databázového modelu prebieha mapovaním entitít a ich vzájomných vzťahov. [27]

Prácu s databázou zabezpečuje trieda *DatabaseManager*. Pôvodné metódy boli nahradené ich generickou alternatívou. Ďalej boli operácie nad databázovými entitami presunuté do samostatných tried.

3.4.2 RobotServerCommunication

Táto trieda zabezpečuje komunikáciu so serverom pre konkrétneho robota. Využíva triedu *HttpClient* pre odosielanie HTTP požiadaviek. Získavané dáta sú následne predávané triede *RobotService*.

3.4.3 RobotService

Jedná sa o triedu, ktorá si udržiava zoznam komunikujúcich robotov, spracováva získavané dáta a obsahuje operácie pre začatie, aktualizáciu alebo ukončenie komunikácie. Ďalej inicializuje SignalR pripojenie prostredníctvom triedy *SignalRService*.

3.4.4 RobotSynchService

RobotSynchService zabezpečuje synchronizáciu skupín a robotov medzi zariadením a serverom. Synchronizáciu vykonáva metóda s názvom *synchronize* a spúšťajú ju nasledujúce udalosti:

- Aktualizácia prihlasovacích údajov.
- Manuálne spustenie užívateľom.
- Aktualizácia na pozadí.

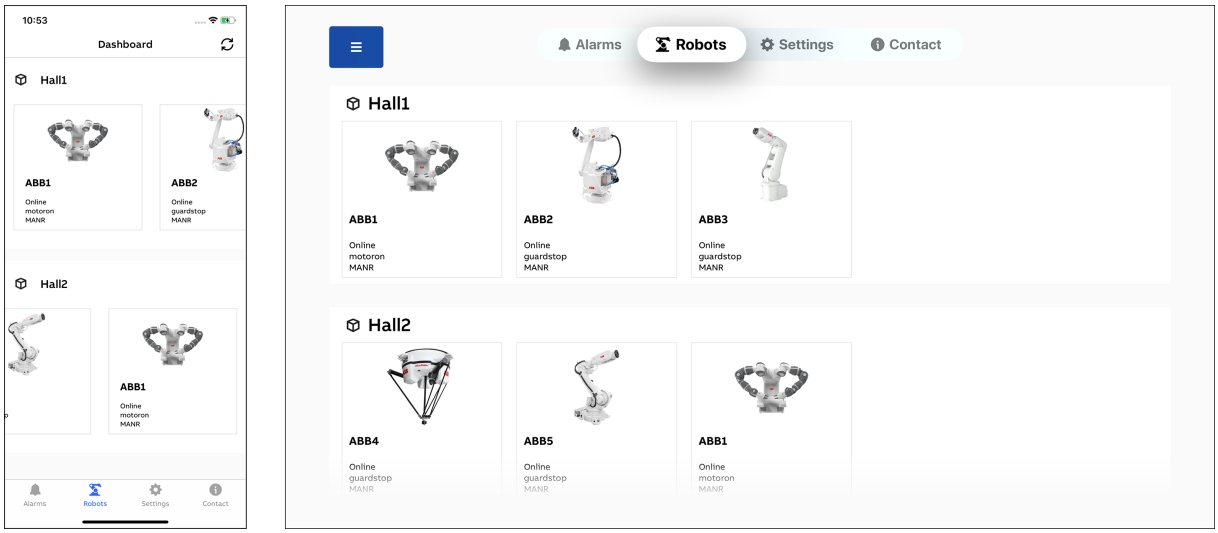
Ukážka metódy *synchronize* je zobrazená vo Výpise 16.

```
func synchronize(handler: @escaping (Bool) -> Void) {
    HttpClient.getInstance().getItems(method: UrlConstant.groupsWithRobots,
        itemType: ServerGroup.self){ serverGroups, isSuccess in
        guard isSuccess else {
            handler(false)
            return
        }

        let groups = self.groupDataAccess.getAll()
        let robots = groups.flatMap({$0.robots})
        let serverRobots = serverGroups.flatMap{x in x.robots}

        let groupsToDelete = groups.filter{ x in !serverGroups.contains(where:
            {x.id == $0.id})}
        groupsToDelete.forEach{groupTable in self.groupDataAccess.delete(id:
            groupTable.id) }

        let robotCommunicationsToDelete = robots.filter{x in !serverRobots.
            contains(where: {y in y.id == x.id})}
        robotCommunicationsToDelete.forEach{r in self.robotService.delete(robot
            : RobotConvert.robotTableToRobot(robotTable: r))}

        self.addOrUpdateGroupsWithRobots(serverGroups: serverGroups, robots:
            robots)

        let robotsToDelete = self.robotDataAccess.getAll().filter{x in !
            serverRobots.contains(where: {y in y.id == x.id})}
        robotsToDelete.forEach{robot in self.robotDataAccess.delete(id: robot.
            id) }

        RobotService.getInstance().events.trigger(eventName: StringConstant.
            eventGroupRobotsSynch)
        handler(true)
    }
}
```

Výpis 16: Synchronizácia skupín a robotov

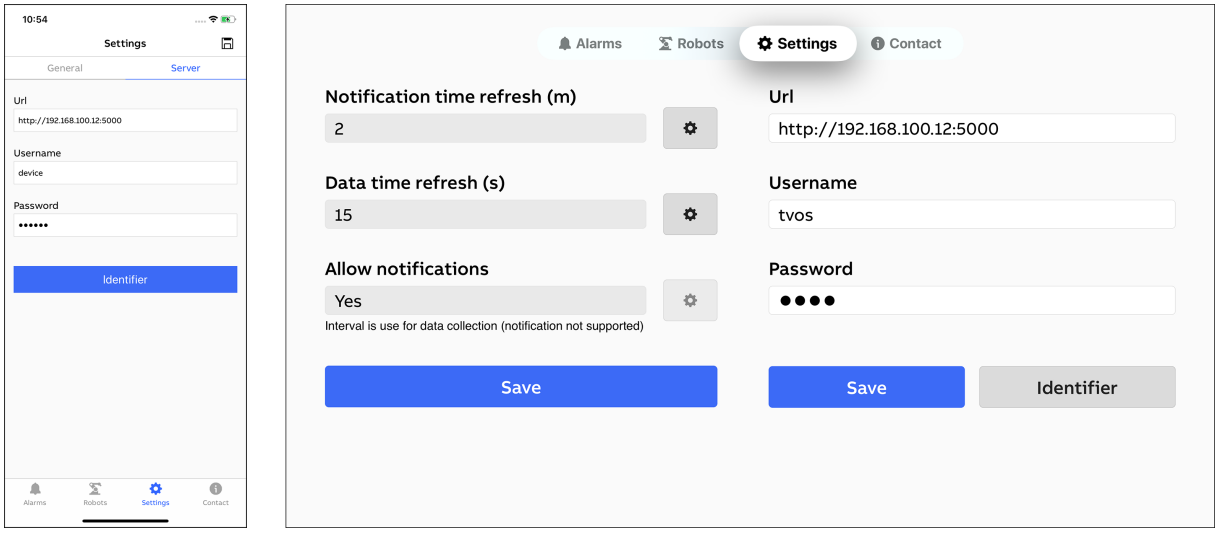
3.4.5 Užívateľské rozhranie

Užívateľské rozhranie si prešlo menšími zmenami. Na obrazovke *Dashboard* došlo k nahradeniu tlačidla pre pridanie robota za tlačidlo spustenia synchronizácie, viď Obrázok 36. Na obrázku môžete ďalej vidieť, že robot s názvom ABB1 je súčasťou viacerých skupín.



Obr. 36: iOS & tvOS - Dashboard úpravy

Ďalšia zmena je viditeľná na obrazovke *Settings*. Došlo k nahradeniu sekcie pre editáciu skupín za nastavenia prihlasovacích údajov k serveru. Táto zmena je zobrazená na Obrázku 37. Ďalej sa tu nachádza tlačidlo s názvom *Identifier*, ktoré užívateľovi zobrazí jedinečný identifikátor zariadenia. Tento identifikátor je potrebné na strane servera spárovať s daným užívateľom.



Obr. 37: iOS & tvOS - Settings úpravy

4 Testovanie

Táto kapitola sa zaoberá testovaním výslednej aplikácie. Popisuje vybrané spôsoby testovania spolu s použitými nástrojmi a knižnicami.

4.1 Unit testovanie

Unit testovanie sa zameriava na otestovanie jednotlivých tried a modulov softvéru. To vývojárom umožňuje ľahšiu identifikáciu chýb spôsobenú zmenami v kóde. Princíp spočíva v porovnaní funkcionality modulu s jeho špecifikáciou a preukázať, že modul je v rozpore s touto špecifikáciou. [28]

Aplikácia obsahuje unit testy pre vybrané triedy z doménovej vrstvy. Konkrétne sa jedná o balíček *Providers*.

4.1.1 Coverlet

Jedná sa o knižnicu, ktorá slúži na výpočet pokrytia zdrojového kódu unit testami. [29] Výsledky sú zobrazované v príkazovom riadku a ukladané do súboru s názvom *coverage.opencover.xml*. Tento súbor je možné importovať do nástrojov ako ReportGenerator, SonarQube a zobraziť si podrobnejšie informácie o výsledkoch testovania. Coverlet poskytuje nasledujúce typy pokrytí:

- **Line:** Koľko riadkov kódu je pokrytých testami.
- **Branch:** Koľko rôznych priechodov kódu je pokrytých testami.
- **Method:** Aké množstvo metód aplikácie je pokryté testami.

4.1.2 Moq

Táto knižnica slúži na napodobňovanie správania tried, čo dáva vývojárom možnosť vytvárať unit testy, ktoré nie sú závislé na databáze, súborovom systéme, iných moduloch alebo celých aplikáciach. [30]

V aplikácií je táto knižnica použitá ako náhrada za databázovú vrstvu, čo mi umožňuje spúšťať unit testy bez pripojenia do databázy a taktiež je zakaždým zabezpečená rovnaká sada vstupných dát. Registrácia mockovaných tried prebieha prostredníctvom IoC kontajnera v triede *ContainerConfig*. Ukážka je zobrazená vo Výpise 17.

```
private static void RegisterTypes(ServiceCollection serviceCollention)
{
    serviceCollention.AddSingleton(new MockRobotRepository().GetObject());
    serviceCollention.AddSingleton(new MockUserRepository().GetObject());
}
```

Výpis 17: Unit testovanie - registrácia mockovaných tried do IoC kontajnera

Všetky mockované triedy dedia od abstraktnej triedy s názvom *BaseMock*. Tá obsahuje abstraktnú metódu *Setup*, ktorá u všetkých potomkov vynúti nastavenie počiatočných dát a metód. Ďalej sa tu nachádza metóda s názvom *GetObject*, ktorá vracia inštanciu mockovanej triedy. Výpis 18 zobrazuje ukážku tejto triedy.

```
public abstract class BaseMock<T> where T : class
{
    protected readonly Mock<T> _mock = new Mock<T>();
    protected abstract void Setup();

    protected BaseMock()
    {
        Setup();
    }

    public T GetObject()
    {
        return _mock.Object;
    }
}
```

Výpis 18: Unit testovanie - ukážka triedy BaseMock

Na záver je ešte potrebné vykonať samotný mock metód z jednotlivých rozhraní pre prístup k dátam. Vo Výpise 19 je zobrazený mock metódy *Insert* z rozhrania *IRobotRepository*.

```
private void _SetupInsert()
{
    _mock.Setup(x => x.Insert(It.IsAny<Robot>())).Returns((Robot robot) =>
    {
        if (robot.IsNull()) return null;

        robot.Id = Guid.NewGuid();
        _robots.Add(robot);
        return robot.Id;
    });
}
```

Výpis 19: Unit testovanie - ukážka mocku metódy Insert z rozhrania IRobotRepository

4.1.3 NUnit

NUnit predstavuje framework, ktorý umožňuje písanie a beh testov v prostredí .NET. Obsahuje podporu pre parametrizované testy, umožňuje vykonať operácie pred začatím alebo po ukončení, poprípade nastaviť ich poradie a iné. [31]

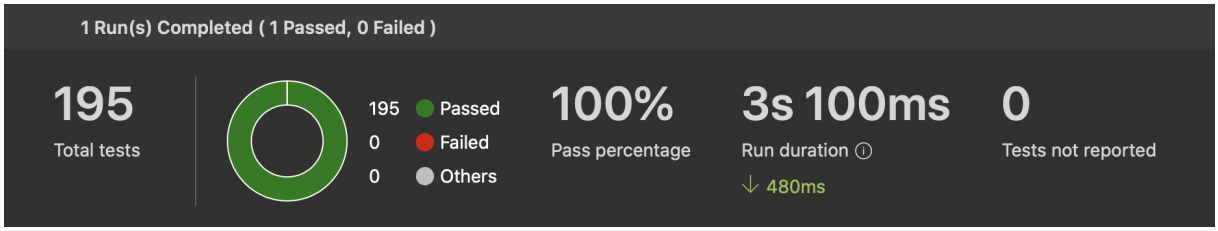
Aby NUnit dokázal identifikovať testy, musia byť triedy, ktoré ich obsahujú, označené atribútom [TestFixture] a následne jednotlivé metódy ako [Test]. Základ frameworku tvorí trieda *Assert*, ktorá obsahuje širokú škálu metód pre vyhodnocovanie tvrdení. V prípade nepravdivého tvrdenia dochádza k vyhodeniu výnimky. Ukážka parametrizovaného testu pre úspešné vloženie robota je zobrazená vo Výpise 20.

```
[Test]
[TestCaseSource(typeof(RobotProviderTestData), "InsertSuccess")]
public void Test_RobotProvider_Insert_Success(RobotDTO robot)
{
    try
    {
        var beforeInsert = _mockRepository.GetRobots();
        Assert.IsNotNull(beforeInsert);
        Assert.IsNotEmpty(beforeInsert);

        var beforeInsertCount = beforeInsert.Count();
        var id = _robotProvider.Insert(robot);
        Assert.IsNotNull(id);
        Assert.IsNotNull(_mockRepository.Get((Guid)id));

        var afterInsert = _mockRepository.GetRobots();
        Assert.IsNotNull(afterInsert);
        Assert.IsNotEmpty(afterInsert);

        Assert.IsTrue(beforeInsertCount < afterInsert.Count());
    }
    catch (Exception e)
    {
        Assert.Fail($"{GetType().Name}.{MethodBase.GetCurrentMethod().Name}
            throw exeption: {e}. Primary exception: {e.GetBaseException()}");
    }
}
```

Výpis 20: Unit testovanie - ukážka testu pre úspešné vloženie robota

4.1.4 Automatické vykonávanie testov

Spúšťanie testov je vykonávané prostredníctvom platformy *Azure DevOps*, viď Sekcia 5.3. Po úspešnom nahratí zmien dochádza k automatickému buildu aplikácie, ktorého súčasťou je beh unit testov. V prípade zlyhania niektorého z testov je build zastavený a užívateľ notifikovaný emailom. Ďalej sa tu nachádza prehľad všetkých vykonaných testov, viď Obrázok 38.



Obr. 38: Azure DevOps - výsledky unit testovania

4.2 Výkonnostné testovanie

Tento typ testovania sa používa pre určenie správania systému pod určitou záťažou. Slúži na identifikovanie úzkych miest a pomáha nám overiť, či aplikácia spĺňa požadované výkonnostné kritéria. Zameriava sa na parametre ako priepustnosť, doba odozvy alebo množstvo využitých zdrojov servera. [32]

4.2.1 Testovacie prostredie

V pilotnej fáze projektu bude aplikácia pracovať s malým počtom užívateľov a robotov, avšak postupom času sa plánuje zavádzanie do väčších podnikov a firiem. Z tohoto dôvodu bolo výkonnostné testovanie vykonané voči dvom typom serverov s rôznou úrovňou výkonu. Konfigurácie serverov sú uvedené v Tabuľke 5.

Tabuľka 5: Technické špecifikácie testovacích serverov

	Ubuntu Server	Raspbian Server
CPU	4 x 2.7 Ghz amd64	4 x 1.4 GHz arm32
RAM	8 GB DDR4	1 GB LPDDR2
Disk	256 GB SSD - 3500/3000 MB/s	32 GB SD karta - 90/50 MB/s
Sieť	Gigabit Ethernet	Gigabit Ethernet - max 300 Mb/s
OS	Ubuntu Server 18.04.4	Raspbian Buster Stretch 9.8
Docker	19.03.8	19.03.8

Výkonnostné testovanie pracuje s pojmom transakcia. Jedná sa o typickú aktivitu užívateľa so serverom. Pri testovaní boli použité nasledujúce typy transakcií:

- **Dashboard, Detail:** Prehliadanie robotov a zobrazovanie ich detailov.
- **Alarm, Data, Status:** Zobrazenie a filtrovanie dát, alarmov a statusov.
- **Administration:** CRUD operácie nad robotmi, užívateľmi, skupinami a koncovými bodmi.
- **Device:** Získanie skupín, robotov a ich údajov pre iOS a tvOS aplikácie.

Každá transakcia navyše zahŕňa autentifikáciu užívateľa.

Dôležitú úlohu pri výkonnostnom testovaní zohrávajú dáta. Aby boli výsledky relevantné, je potrebné mať dáta v dostatočnej kvantite a kvalite. Testovanie bolo vykonané na vzorke, ktorá odpovedá trojtýždňovému zberu dát z 5 robotov. V Tabuľke 6 sú uvedené počty zozbieraných údajov.

Tabuľka 6: Výkonnostné testovanie - počty testovacích dát

	Počet
Data	1 532 555
Alarm	3 813
Status	252 903

Aby som sa čo najviac priblížil reálnemu používaniu, bola počas testovania povolená komunikácia s 5 robotmi prostredníctvom WebSocket a HTTP protokolu. Po každom testovaní bol vykonaný reštart servera a následná obnova databázy.

4.2.2 JMeter

Jedná sa o open source softvér napísaný v jazyku Java. Tento nástroj slúži na testovanie funkčného správania systému a meranie jeho výkonu. Poskytuje širokú škálu konfigurácie testov, vizualizáciu výsledkov a inštaláciu rozšírení. [33]

JMeter obsahuje rozličné druhy elementov, pomocou ktorých je možné zostaviť testovací plán. Základný prvok predstavuje *Thread Groupa*, ktorá v závislosti na použitom type umožňuje nastaviť počet vlákien, dobu behu, postupné navyšovanie alebo znižovanie záťaže. Pri testovaní som použil *Ultimate Thread Groupy*, ktoré poskytujú najvyššiu mieru konfigurácie.

Ďalej sa tu nachádzajú *Controllery*, ktoré riadia priebeh testu. Umožňujú odosielať požiadavky v náhodnom poradí, cykle, na základe podmienky a podobne. Pomocou Controllerov logicky oddeľujem jednotlivé aktivity užívateľa so serverom.

O to kam a aké (HTTP, FTP, ...) požiadavky budu odoslané sa stará prvok s názvom *Sampler*. V mojom prípade sú všetky požiadavky typu HTTP. V Tabuľke 7 sú uvedené počty požiadaviek v jednotlivých transakciách.

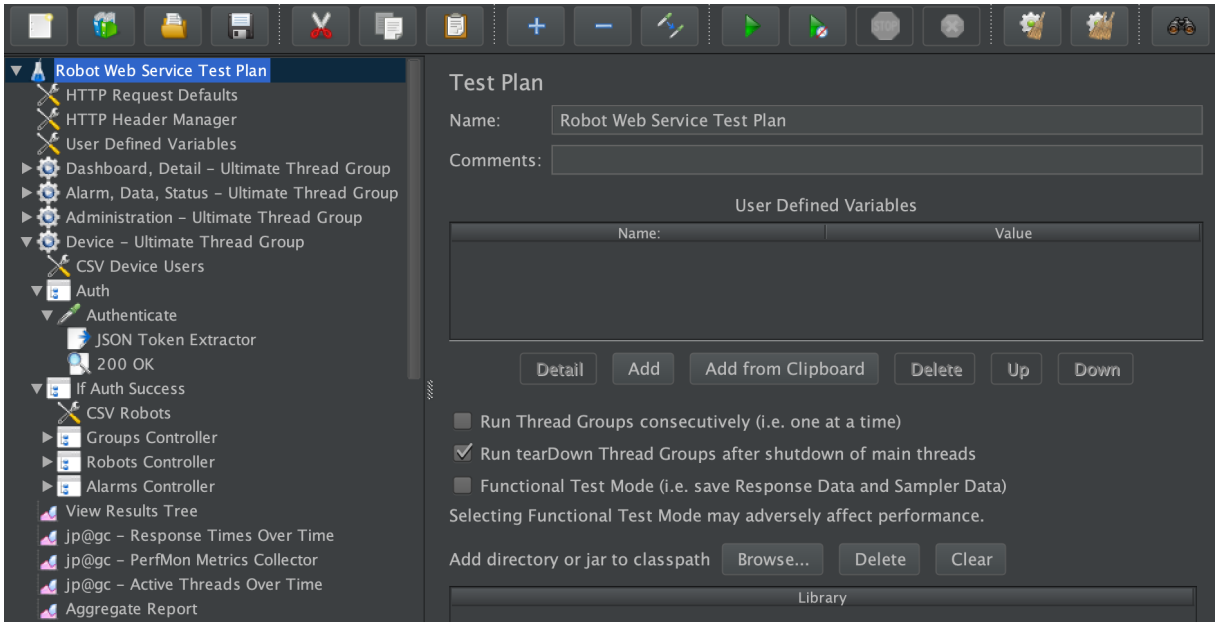
Tabuľka 7: Výkonnostné testovanie - počty HTTP požiadaviek v transakciách

	Počet
Dashboard, Detail	20
Alarm, Data, Status	10
Administration	27
Device	20
Súčet	**77**

Dáta získané z jednotlivých testov je možné zobrazovať, ukladať alebo odosielať do iných aplikácií prostredníctvom elementu *Listener*. Počas testovania boli použité nasledujúce Listenery:

- **View Result Tree:** Zobrazenie podrobností o jednotlivých HTTP požiadavkách.
- **Response Times Over Time:** Graf doby odozvy pre jednotlivé požiadavky.
- **PerfMon Metrics Collector:** Graf využívania zdrojov servera.
- **Aggregate Report:** Meranie priepustnosti, chybovosti, prenesených dát a odozvy.
- **Response Codes per Second:** Graf kódov odpovedí pre jednotlivé požiadavky.

Ďalej bolo pri testovaní využité načítavanie dát z CSV súborov, náhodné intervaly medzi požiadavkami, kontrola kódu odpovede, premenné a json/regex extraktory. Ukážku testovacieho plánu v nástroji JMeter môžete vidieť na Obrázku 39.



Obr. 39: Ukážka testovacieho plánu v nástroji JMeter

4.2.3 Crankier

Jedná sa o nástroj od spoločnosti Microsoft pre výkonnostné testovanie SignalR komunikácie. Umožňuje nastaviť spôsob prenosu, cieľový počet pripojení, čas trvania alebo počet súbežných klientov. Jeho jedinou úlohou je nadviazanie a udržanie čo najväčšieho počtu spojení s konkrétnym Hubom. [34]

To však moc neodpovedá reálnemu používaniu a preto bola vykonaná menšia úprava kódu, aby bolo možné nasimulovať reálne pripojenie so serverom. Po modifikácií je Crankier schopný nadviazať spojenie so všetkými Hubmi na serveri, podporuje autentifikáciu a v neposlednom rade sa registruje na poskytované metódy a očakáva správy zo servera. Ukážka spustenia testu prostredníctvom príkazového riadku je zobrazená vo Výpise 21.

```
dotnet run -- local --target-url http://172.20.10.8 --dashboardhub 100
    --mobilehub 100 --customdatahub 100
```

Výpis 21: Spustenie nástroja Crankier

Tento príkaz zabezpečí otestovanie servera na adrese *http://172.20.10.8*, pričom sa snaží nadviazať 100 pripojení s každým Hubom. Počas testovania dochádza k zberu rôznych štatistík komunikácie (aktuálny počet pripojení, maximálny počet pripojení, počet zlyhaní).

4.2.4 SignalR testovanie

Keďže SignalR predstavuje jeden z hlavných aspektov aplikácie, je dôležité poznať jeho vplyv na výkon systému. Pri tomto type pripojenia dochádza k nadviazaniu trvalého obojstranného pripojenia medzi klientom a serverom, čo má za následok určitú spotrebu zdrojov servera.

Cieľom tohoto testovania bolo zistiť maximálny počet pripojení, ktoré dokáže server obslúžiť a taktiež vplyv SignalR komunikácie na využívanie jeho zdrojov. Aby bolo možné nadviazať čo najväčší počet spojení, tak došlo k úprave konfigurácie SignalR protokolu. Jednalo sa o nasledujúce parametre:

- **ClientTimeoutInterval:** Pokiaľ server neobdrží od klienta správu v tomto intervale, považuje ho za odpojený.
- **KeepAliveInterval:** Ak server neodoslal správu v tomto intervale, automaticky sa odošle správa ping, aby sa pripojenie udržalo otvorené.
- **ServerTimeout:** Pokiaľ klient neobdrží od servera správu v tomto intervale, považuje ho za odpojený.

V Tabuľke 8 sú uvedené predvolené a nové hodnoty konfigurácie pre SignalR komunikáciu.

Tabuľka 8: SignalR testovanie - konfigurácia

	Predvolené hodnoty (sekundy)	Nové hodnoty (sekundy)
ClientTimeoutInterval	30	120
KeepAliveInterval	15	60
ServerTimeout	30	120

Počas testovania som narazil na problém, pri ktorom nezávisle na type testovaného serveru nebolo možné nadviazať od určitého bodu nové spojenia. Tento problém bol spôsobený vyčerpaním lokálneho rozsahu portov na mojom testovacom zariadení. V prípade operačného systému *macOS* je riešením príkaz *sysctl -w net.inet.ip.portrange.first="pociatocny_port"*, ktorý nastaví rozsah od počiatočného portu po hodnotu 65 535. Pre potreby testovania som si vystačil s hodnotou 35 535, čo mi dalo k dispozícií celkovo 30 000 portov. Na každom serveri boli vykonané 3 merania s dĺžkou 30 minút.

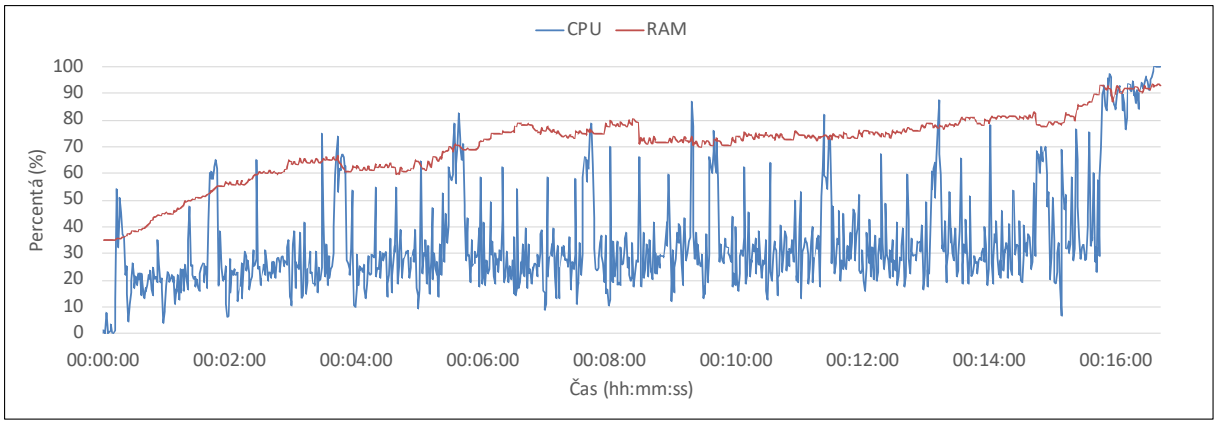
Raspbian Server

V Tabuľke 9 sú uvedené namerané hodnoty výkonnostného testovania SignalR komunikácie pre Raspbian Server. Každé z meraní obsahuje vysoký počet zlyhaní, ktoré boli spôsobené vyčerpaním zdrojov servera a jeho úplnou nedostupnosťou. Do tohoto bodu prebiehala komunikácia bez výraznejších problémov, s výnimkou pár zlyhaní spôsobených vypršaním času pre výmenu informácie medzi klientom a serverom. Do zlyhaní sa započítavali aj hodnoty neúspešného naviazania komunikácie zo strany klienta a nielen prerušenia už existujúcich komunikácií. Z tohoto dôvodu môžu hodnoty v stĺpci Zlyhanie nadobúdať väčších počtov ako v prípade stĺpca Maximum.

Tabuľka 9: SignalR testovanie - výsledky testovania pre Raspbian Server

	Zlyhanie	Maximum
1. meranie	13 688	13 683
2. meranie	14 425	14 221
3. meranie	13 569	13 560
Priemer	**13 894**	**13 821**

Na Obrázku 40 je zobrazený graf vyťaženia servera počas prvého merania. Na grafe možno pozorovať, že so zvyšujúcou sa zátažou dochádzalo k postupnému nárastu využívania pamäte RAM až sa úplne vyčerpala. Pri CPU bol nárast podstatne pomalší, avšak s blížiacim sa bodom vyčerpania pamäte došlo k jeho skoku k hranici 100%. Vyťaženie CPU a RAM pri zvyšných meraniach malo obdobný priebeh.



Obr. 40: SignalR testovanie - graf vyťaženia pre Raspbian Server

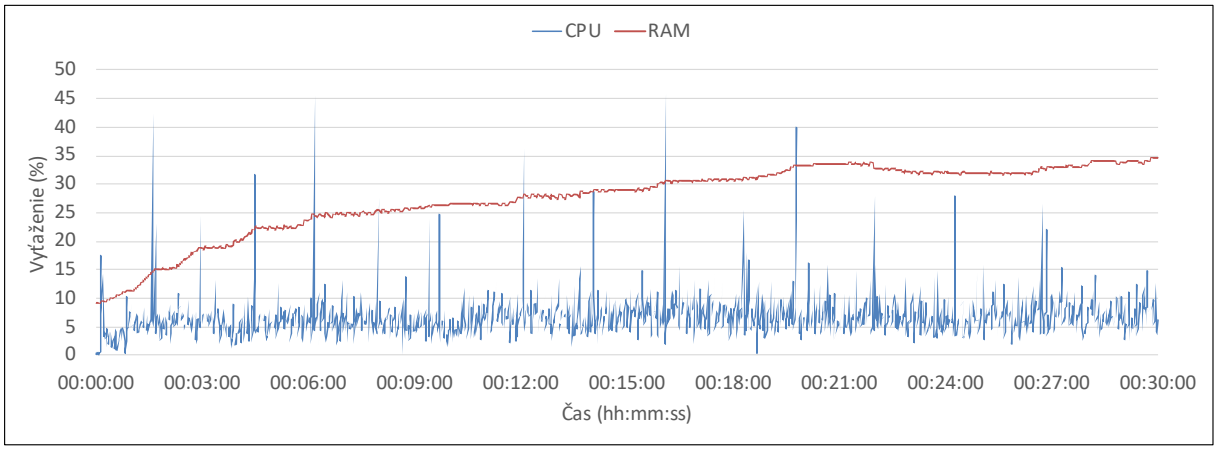
Ubuntu Server

Výsledky meraní pre Ubuntu Server môžete vidieť v Tabuľke 10. Oproti Raspbian Serveru došlo s nárastu maximálneho počtu pripojení a zníženiu počtu zlyhaní. Jedná sa o očakávaný výsledok vzhľadom k výkonnejšiemu hardvéru. Zlyhania spôsobila neschopnosť klienta so serverom vymeniť si informácie do definovaného časového obdobia.

Tabuľka 10: SignalR testovanie - výsledky testovania pre Ubuntu Server

	Zlyhanie	Maximum
1. meranie	5 584	24 772
2. meranie	7 109	25 317
3. meranie	4 126	23 549
Priemer	**5 606**	**24 546**

Graf na Obrázku 41 zobrazuje priebeh využívania zdrojov servera počas prvého merania.



Obr. 41: SignalR testovanie - graf vyťaženia pre Ubuntu Server

Na grafe možno pozorovať nárast vyťaženia pamäte RAM z 10% na 35%. Zvyšujúca sa záťaž mala minimálny vplyv na CPU a ten sa stabilne pohyboval medzi 5% až 10%, s výnimkou občasných výkyvov nad alebo pod tento rozsah. Zvyšné merania vyťažili server podobným spôsobom.

4.2.5 Load testovanie

Jedná sa o typ testovania, pri ktorom dochádza k sledovaniu aplikácie pri požadovanej úrovni záťaže. Cieľom je splniť definované výkonnostné kritéria. Load testovanie predstavuje najbližšiu aproximáciu skutočného používania.

Pri testovaní bola použitá metóda *Big Bang*, ktorá spúšťa všetkých užívateľov súčasne, čím aplikuje požadovanú záťaž od začiatku až po koniec testovania. Predpokladané hodnoty užívateľov pre oba servery sú uvedené v tabuľke 11. Počas testovania bol taktiež sledovaný vplyv záťaže na stabilitu SignalR komunikácie. Každé testovanie prebiehalo po dobu 10 minút.

Tabuľka 11: Load testovanie - predpokladané hodnoty záťaže

	Raspbian Server		Ubuntu Server	
	Užívateľ	SignalR	Užívateľ	SignalR
Dashboard, Detail	2	2	15	15
Alarm, Data, Status	2	2	15	15
Administration	1	0	5	0
Device	5	5	15	15
Súčet	**10**	**9**	**50**	**45**

Raspbian Server

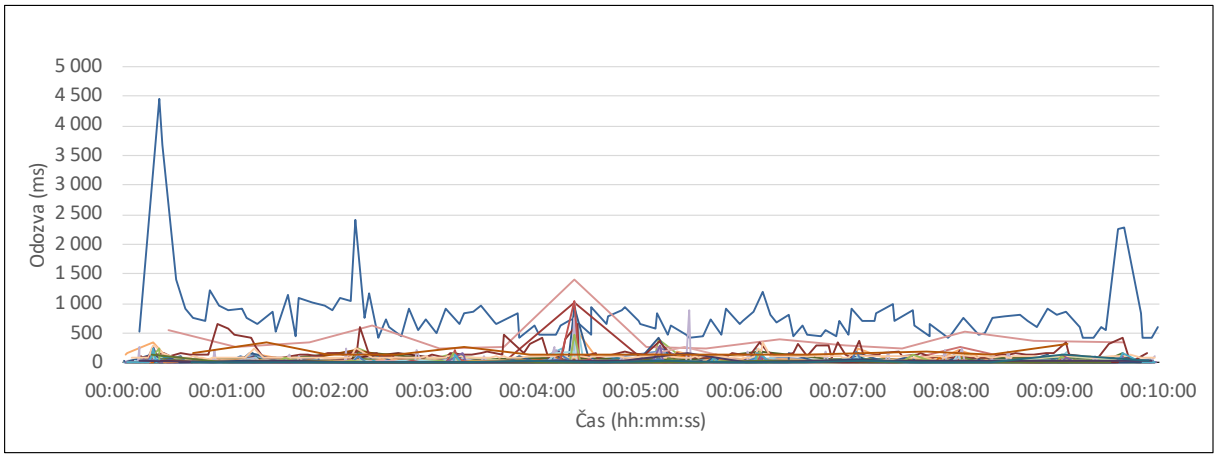
Namerané hodnoty testovania sú uvedené v Tabuľke 12. Počas 10 minút bolo na server odoslaných celkovo 4 956 požiadaviek a všetky prebehli úspešne. Priemerná doba vybavenia požiadavky bola 78 ms. Ďalší dôležitý údaj nám poskytuje percentil, ktorý udáva koľko percent požiadaviek bolo rýchlejších ako daná hodnota. Tu možno vidieť, že až 99% požiadaviek bolo spracovaných do 1 sekundy. Počas merania nedošlo k zlyhaniu žiadneho SignalR pripojenia.

Tabuľka 12: Load testovanie - výsledok testovania pre Raspbian Server

Počet	Priemer	Chyby	Priepustnosť	Percentil (ms)			
				50%	90%	95%	99%
4 956	78 ms	0%	8.3 požiadavka/s	20	141	430	941

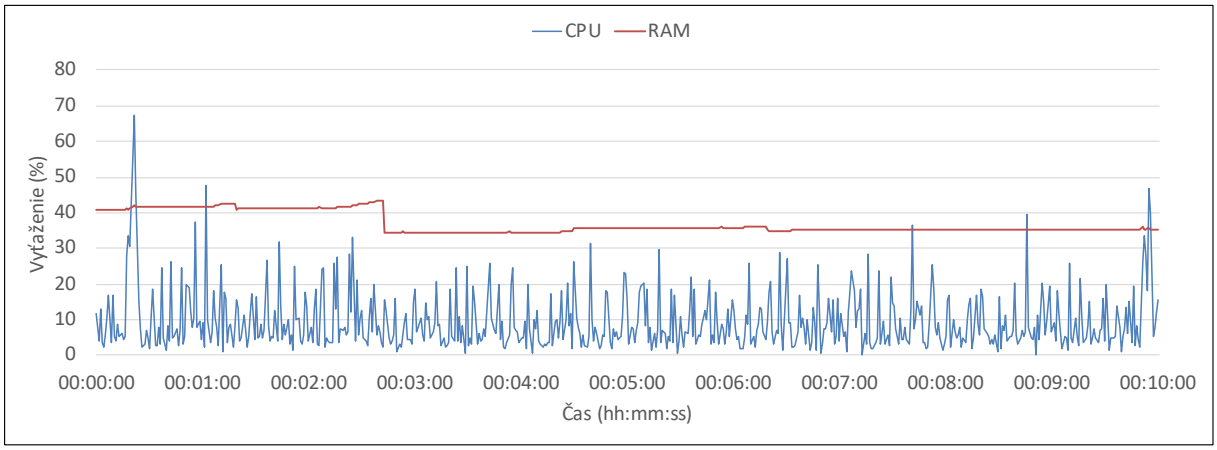
Na Obrázku 42 je zobrazený graf odozvy pre jednotlivé požiadavky, na ktorom možno vidieť, že čas odpovedí sa prevažne pohyboval v desiatkach až stovkách milisekúnd. V prípade krivky atakujúcej hranicu 4.5 sekundy sa jednalo o požiadavku pre získanie dát z robotov v rozsahu niekoľkých dní, čo predstavovalo až 350 000 záznamov naraz. Pre porovnanie, veľkosti odpovedí

sa pri zvyšných požiadavkách pohybovali na úrovni jednotiek alebo desiatok záznamov.



Obr. 42: Load testovanie - graf odozvy pre Raspbian Server

Graf na Obrázku 43 zobrazuje priebeh vyťaženia servera. Procesor ako aj pamäť RAM sa držali v nízkych hodnotách počas celej doby testovania.



Obr. 43: Load testovanie - graf vyťaženia pre Raspbian Server

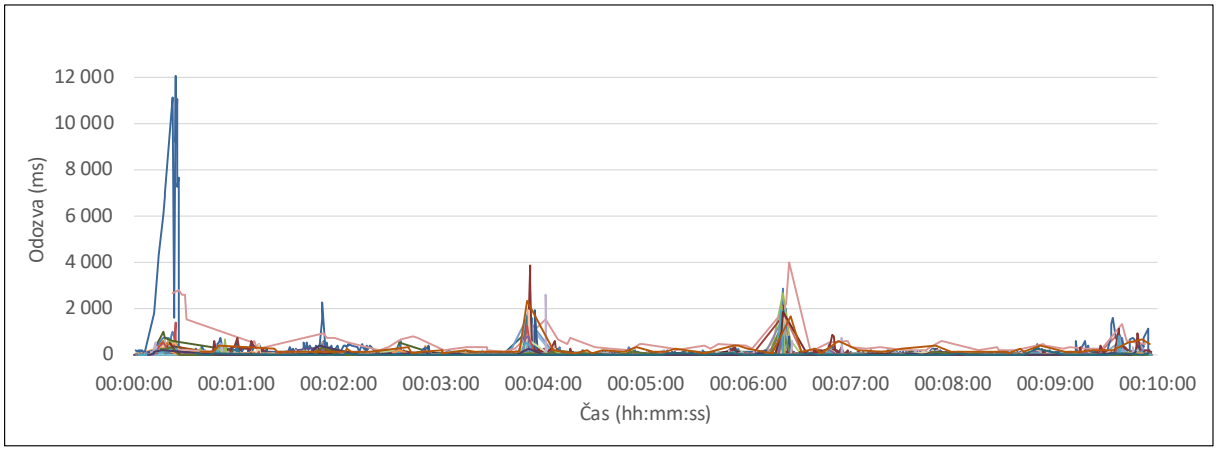
Ubuntu Server

V Tabuľke 13 možno vidieť, že celkový počet odoslaných požiadaviek sa vyšplhal na hodnotu 28 244. Chybovosť týchto požiadaviek ako aj SignalR pripojení bola 0%. Server zvládal záťaž bez problémov, o čom svedčí aj priemerná doba odozvy alebo namerané hodnoty percentilov.

Tabuľka 13: Load testovanie - výsledok testovania pre Ubuntu Server

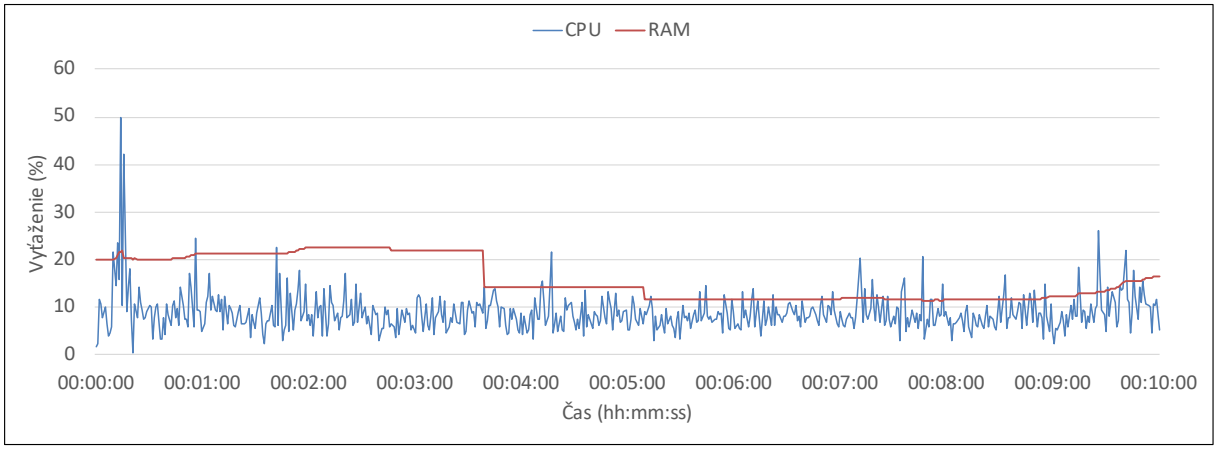
Počet	Priemer	Chyby	Priepustnosť	Percentil (ms)			
				50%	90%	95%	99%
28 244	65 ms	0%	47 požiadavka/s	15	126	221	747

Na grafe z Obrázka 44 možno pozorovať, že až na väčší výkyv zo začiatku merania, kedy server v stave nečinnosti musel nárazovo spracovať desiatky požiadaviek naraz, sa doba odozvy väčšinu času pohybovala pod úrovňou 1 sekundy. Najväčší problém opäť predstavovali požiadavky pre získanie dát z robotov.



Obr. 44: Load testovanie - graf odozvy pre Ubuntu Server

Výsledný graf vyťaženia servera je uvedený na Obrázku 45. Možno tu vidieť, že hodnoty CPU sa pohybovali prevažne v okolí 10% a využívanie pamäte RAM bolo v rozsahu 10% až 25%.



Obr. 45: Load testovanie - graf vyťaženia pre Ubuntu Server

4.2.6 Stress testovanie

Stress testovanie sa narozdiel od load testovania snaží nájsť horné limity aplikácie. Dochádza tu k neustálemu zvyšovaniu záťaže až pokiaľ sa niečo nepokazí. Zvyčajne sa jedná o vysokú dobu odozvy a neschopnosť spracovávať nové požiadavky. Zameriava sa na meranie kapacity a výkonu, pretože je dôležité poznať limity aplikácie, hlavne ak je ťažké predvídať budúci nárast prevádzky.

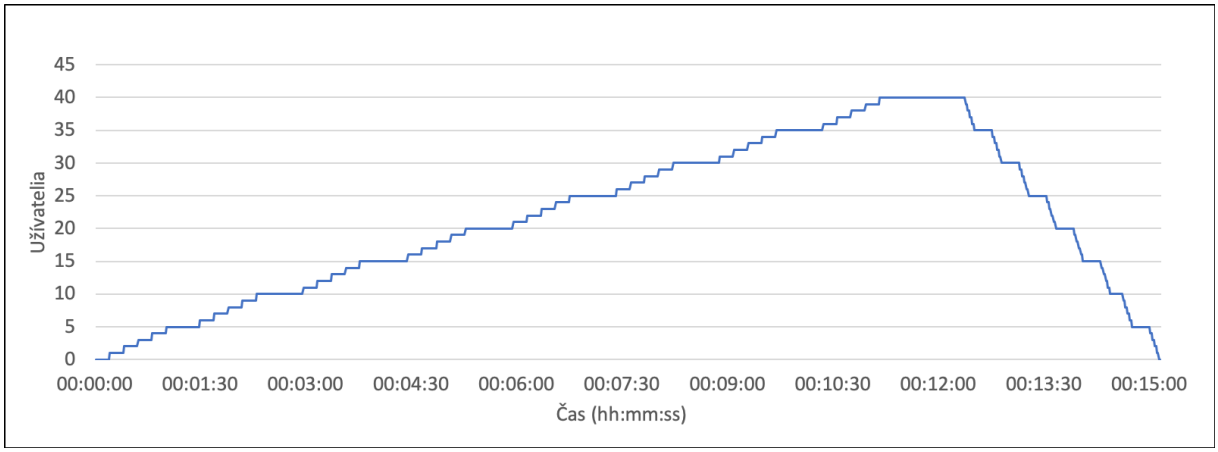
Pri tomto type testovania bola zvolená metóda *Ramp-up*. Jej princíp spočíva v postupnom navyšovaní počtu užívateľov v definovaných intervaloch až do požadovanej hodnoty. Cieľový počet užívateľov a SignalR pripojení pre oba servery je uvedený v Tabuľke 14. Tentokrát bola záťaž rovnomerne rozdelená medzi jednotlivé transakcie. Počas testovania som sledoval nielen vplyv veľkého množstva užívateľov na výkon, ale aj schopnosť servera zotaviť sa pri klesajúcej záťaži. Dĺžka testovania bola 15 minút.

Tabuľka 14: Stress testovanie - maximálne hodnoty záťaže

	Užívateľ	SignalR
Raspbian Server	160	120
Ubuntu Server	800	600

Raspbian Server

Na Obrázku 46 je zobrazený priebeh navyšovania záťaže pre každú z transakcií. V 90-sekundových intervaloch dochádza k nárastu počtu užívateľov o 20 až po požadovaný počet 160. Následne je maximálna záťaž udržiavaná po dobu 75 sekúnd, po ktorých prichádza jej postupné znižovanie.



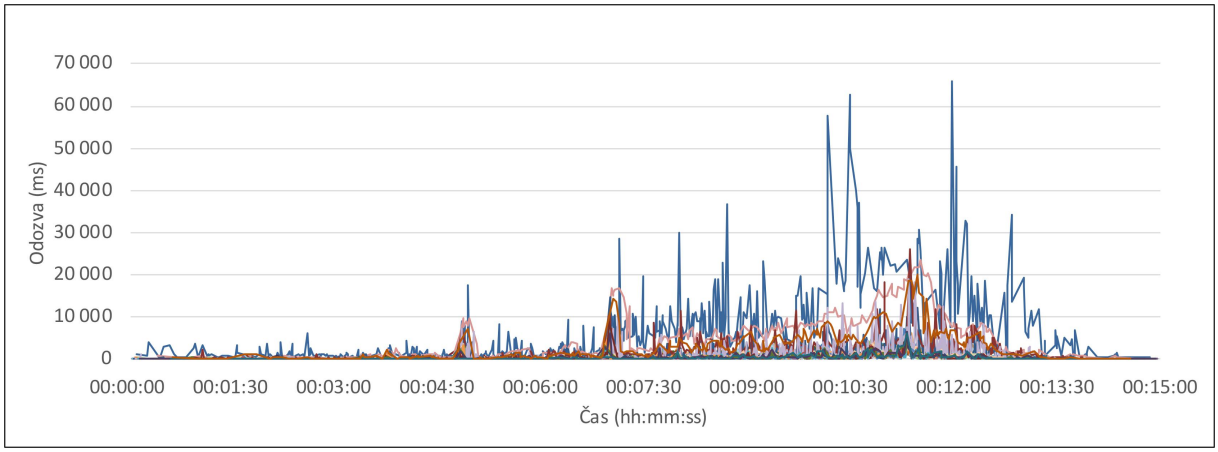
Obr. 46: Stress testovanie - priebeh navyšovania záťaže pre Raspbian Server

Výsledky testovania môžete vidieť v Tabuľke 15. Aj napriek zvýšenej záťaži bol server schopný spracovať 90% požiadaviek do 2 sekúnd. Výsledná priepustnosť činila 54.2 požiadavky za sekundu.

Tabuľka 15: Stress testovanie - výsledok testovania pre Raspbian Server

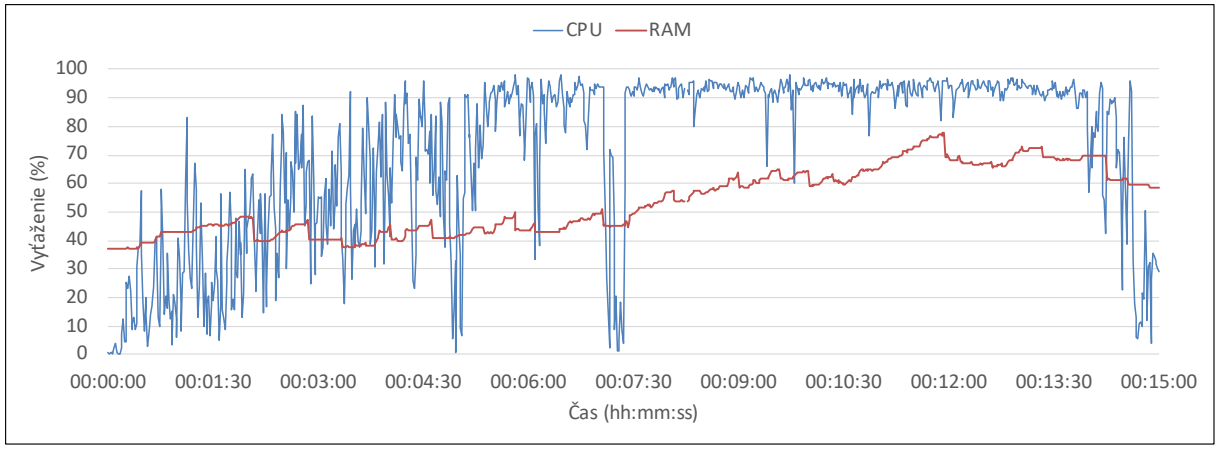
Počet	Priemer	Chyby	Priepustnosť	Percentil (ms)			
				50%	90%	95%	99%
48 079	915 ms	0%	54.2 požiadavka/s	239	1 891	3 394	12 959

Graf na Obrázku 47 nám dáva dobrú predstavu o priebehu spracovávania jednotlivých požiadaviek so zvyšujúcou sa záťažou. Na grafe možno vidieť, že server nemal zo začiatku merania problém so spracovávaním požiadaviek a odozva sa pohybovala v rozsahu jednotiek sekúnd. Od hranice 100 užívateľov súčasne začala doba odpovedí postupne narastať a pri maximálnej záťaži atakovala až 70-sekundovú hranicu. Aj napriek tomu bol server schopný udržať všetky SignalR spojenia a neodoslať žiadnu chybnú odpoveď. Ďalej možno vidieť ako s klesajúcim počtom užívateľov dochádzalo k rýchlejšiemu spracovávaniu požiadaviek.



Obr. 47: Stress testovanie - graf odozvy pre Raspbian Server

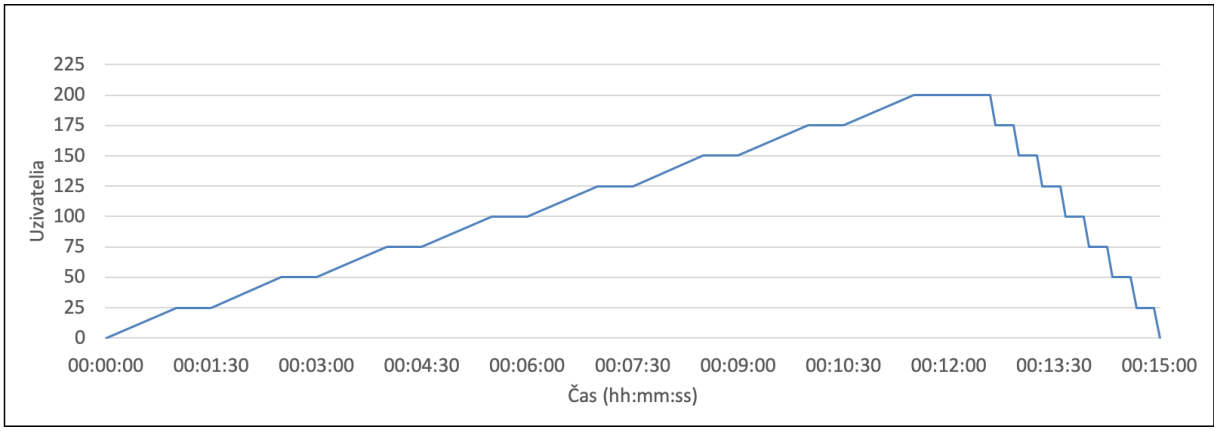
Priebeh vyťaženia zdrojov servera je zobrazený na Obrázku 48. Od záťaže 80 užívateľov súčasne sa vyťaženie CPU pohybovalo na úrovni 100%. Pamäť RAM pri maximálnej záťaži atakovala hodnotu 80%. Pri ďalšom zvyšovaní počtu užívateľov by došlo k jej vyčerpaniu. Klesajúca záťaž na konci merania spôsobila okamžité zníženie vyťaženia sledovaných zdrojov.



Obr. 48: Stress testovanie - graf vyťaženia pre Raspbian Server

Ubuntu Server

Pri testovaní tohoto typu servera bol zvolený podobný model ako v prípade Raspbian Servera, viď Obrázok 49. Adekvátne k nárastu výkonu sa navýšil aj počet užívateľov, ktorí pribúdajú v 90-sekundových intervaloch, na hodnotu 100. To predstavuje 25 užívateľov na transakciu.



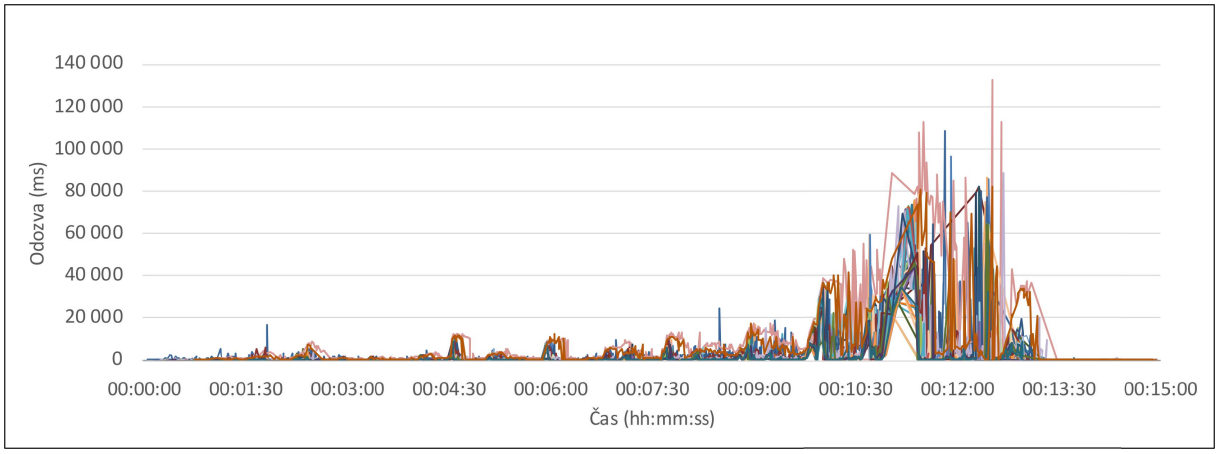
Obr. 49: Stress testovanie - priebeh navyšovania záťaže pre Ubuntu Server

V Tabuľke 16 sú uvedené namerané hodnoty stress testovania. Počas 15 minút dokázal server spracovať celkovo 179 166 požiadaviek s chybovosťou ≈4%, čo predstavuje niečo cez 7 100 chybných odpovedí. Problémom sa nevyhla ani SignalR komunikácia, ktorá zlyhala v 201 prípadoch zo 600 a to z dôvodu neschopnosti klienta a servera vymeniť si potrebné správy pre jej udržanie. Všetky spomenuté problémy sa začali objavovať na úrovni 400 užívateľov súčasne. Na druhú stranu však server dosiahol obrovskú priepustnosť, čo potvrdzujú aj namerané hodnoty pre percentil alebo priemerná doba odozvy.

Tabuľka 16: Stress testovanie - výsledok testovania pre Ubuntu Server

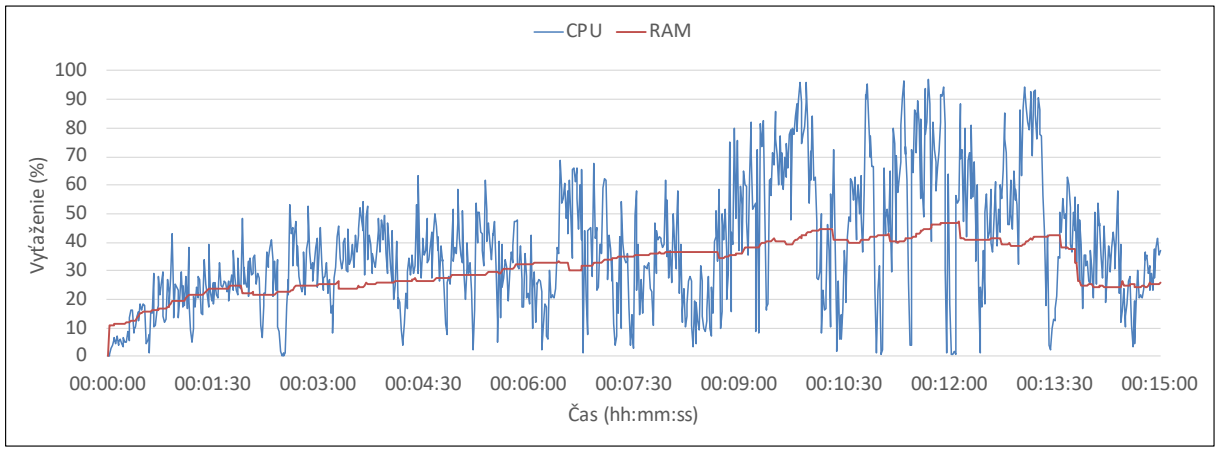
Počet	Priemer	Chyby	Priepustnosť	Percentil (ms)			
				50%	90%	95%	99%
179 166	1 349 ms	3.972%	199.6 požiadavka/s	165	2 205	6 003	26 372

Na Obrázku 50 je zobrazený graf doby odozvy pre jednotlivé požiadavky. Možno z neho vyčítať, že server si v prvej polovici merania (600 užívateľov) počínal veľmi dobre. Od tohoto bodu však dochádzalo k postupnej degradácií výkonu a nárastu počtu chýb. Pri maximálnej záťaži sa odpovede bežne pohybovali nad 1 minútou. Aj napriek tomu ostal server stále dostupný a pri znížení záťaže sa spracovanie odpovedí okamžite dostalo na hodnoty zo začiatku merania.



Obr. 50: Stress testovanie - graf odozvy pre Ubuntu Server

Vyťaženie CPU a RAM, ktoré môžete vidieť na Obrázku 51 sa pohybovalo v prijateľných hodnotách. Pamäť pri maximálnej záťaži atakovala hodnotu 50% a procesor sa občas dostal k hranici 100%. Opäť možno vidieť okamžitý pokles vyťaženia na konci merania, ktorý spôsobil ubúdajúci počet užívateľov.



Obr. 51: Stress testovanie - graf vyťaženia pre Ubuntu Server

4.2.7 Zhodnotenie

Výsledky SignalR testovania ukázali, že oba servery boli schopné nadviazať veľké množstvo spojení, ktoré ďaleko presahujú predpokladané hodnoty záťaže, uvedené v tabuľke 11. Toto testovanie mi dalo dobrú predstavu o tom, ako vplýva zvyšujúci sa počet SignalR spojení na zdroje servera. Nárast bol pozorovateľný hlavne na vyťažení pamäte RAM, o čom svedčia aj grafy na Obrázkoch 40 a 41. Jedná sa o predpokladaný výsledok, vzhľadom k tomu, že si server musel udržiavať v pamäti zoznam pripojených klientov. Vyťaženie procesora pri tomto spôsobe testovania vykazovalo len mierny nárast.

Load testovanie mi pomohlo overiť, že oba servery dokážu zvládnuť predpokladaný počet užívateľov, uvedených v Tabuľke 11, bez väčších problémov. Toto tvrdenie potvrdzujú namerané hodnoty uvedené v Tabuľkách 12 a 13. Chybovosť jednotlivých požiadaviek ako aj SignalR spojení bola 0%. Priemerná doba odozvy sa udržala pod hodnotou 100 milisekúnd. Dôležité informácie taktiež poskytujú grafy vyťaženia zdrojov pre jednotlivé servery na Obrázkoch 43 a 45. CPU ako aj pamäť RAM sa v oboch prípadoch pohybovali v nízkych hodnotách, čo len dokazuje, že ani jeden zo serverov nepracoval na hrane svojich možností.

Počas stress testovania som pri Raspbian Serveri dospel k veľmi pozitívnym výsledkom. Tie ukázali, že aj tak malý počítač dokáže poskytnúť dostatočný výkon na obslúženie 60 užívateľov, bez toho aby došlo k vyčerpaniu zdrojov servera a s dobou odozvy v rámci stoviek milisekúnd až jednotiek sekúnd, viď Tabuľka 15. Počas celej doby testovania pri tomto type servera nedošlo k odoslaniu chybnej odpovede ani prerušeniu SignalR komunikácie. Zvyšujúca sa záťaž mala vplyv hlavne na vyťaženie CPU, ktoré sa od 100 užívateľov súčasne držalo na úrovni 100%. Ubuntu Server bol taktiež schopný obslúžiť mnohonásobne viac užívateľov ako predpokladaná hodnota, avšak narozdiel od Raspbian Servera boli zaznamenané chybné odpovede a výpadky SignalR spojení. Aj napriek tomu bol však server schopný spracovať 179 166 požiadaviek počas 15 minút s priemernou dobou odozvy 1349 milisekúnd. Ani maximálna záťaž, ktorá činila 800 užívateľov súčasne, nedokázala na plno vyťažiť zdroje servera, viď Obrázok 51.

Výkonnostné testovanie malo v konečnom dôsledku obrovský vplyv na odhalenie problémových častí systému. Prvotné merania už pri predpokladanej úrovni záťaže vykazovali vysokú dobu odozvy a vyťaženie CPU sa pohybovalo na úrovni 100%. Na základe reportov a grafov, ktoré poskytuje nástoj JMeter som identifikoval požiadavky, ktoré trvali najdlhšie a vykonal som následnú analýzu. Tá mi odhalila, že dochádzalo k neoptimálnemu vykonávaniu databázových skriptov. Po ich opravení, ktoré zahŕňalo aj pridanie indexov, sa doba odozvy pre problémové požiadavky znížila z jednotiek sekúnd na stovky milisekúnd. Táto zmena bola okamžite viditeľná aj na základe poklesu vyťaženia CPU. Ďalej mi toto testovanie pomohlo overiť jednotlivé funkcionality systému, od prevolania REST API rozhrania, cez priechod doménovou vrstvou až po získanie dát z databázy. Každý kód odpovede servera bol kontrolovaný JMeterom, čím došlo k otestovaniu predpokladaného výsledku operácie.

4.3 Statická analýza kódu

Pri statickej analýze kódu nedochádza k spusteniu žiadnej časti programu. Môže byť vykonávaná manuálne (postupné prechádzanie kódu iným programátorom) alebo prostredníctvom automatizovaných nástrojov, ktoré na základe širokej škály preddefinovaných vzorov hľadajú chyby v kóde. Týmto spôsobom je možné predchádzať budúcim problémom a zraniteľnostiam. [35]

4.3.1 SonarQube

Jedná sa o open source softvér pre statickú analýzu kódu. Tento nástroj charakterizuje jednoduché použitie, veľké množstvo podporovaných jazykov či spustiteľnosť na rozličných operačných systémoch. Medzi jeho ďalšie výhody patrí Docker podpora, integrácia s nástrojmi pre CI/CD a široká škála doplnkových modulov. [36] SonarQube poskytuje nasledujúcu funkcionalitu:

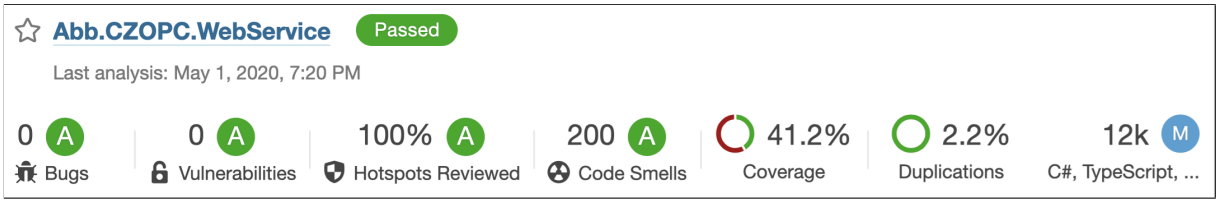
- Zisťovanie chýb, zraniteľností a pachov v kóde (code smells).
- Poskytovanie metrík o kvalite kódu.
- Meranie pokrytia unit testami.
- Informácie o počte riadkov kódu a množstve duplikácií.

4.3.2 Analýza

SonarQube analýza je súčasťou automatického zostavovania aplikácie prostredníctvom platformy Azure DevOps, viď Sekcia 5.3.2. Vo všeobecnosti sa táto analýza skladá z nasledujúcich krokov:

- Vykonanie unit testov a výpočet pokrytia.
- Spustenie analýzy.
- Build aplikácie.
- Ukončenie analýzy a odoslanie výsledkov.

Statická analýza mi pomohla odhaliť niekoľko nedokonalostí v zdrojovom kóde. Jednalo sa prevažne o nevyužité importy knižníc, nezrovnalosti medzi názvami parametrov metód v rozhraní a triede alebo nadbytočné podmienky. Výsledok analýzy môžete vidieť na Obrázku 52.



Obr. 52: Výsledok statickej analýzy kódu

Z výsledku možno pozorovať, že aplikácia obsahuje celkovo 12-tisíc riadkov kódu s minimálnou úrovňou duplikácie. Taktiež neboli detekované žiadne bugy ani zraniteľnosti. Ďalej sa tu nachádza informácia o úrovni pokrytia doménovej vrstvy unit testami. SonarQube udáva výsledok, ktorý kombinuje hodnoty z *Line* a *Branch* pokrytia[13]. Unit testy boli prevažne zamerané na balíček Providers, ktorý dosahuje úroveň pokrytia 84.9%.

[13]Výpočet SonarQube pokrytia: `https://docs.sonarqube.org/latest/user-guide/metric-definitions`

5 Nasadenie

Táto kapitola sa zaoberá procesom nasadenia aplikácie na fyzický server. Popisuje vytvorenie Docker obrazov, postupnosť krokov pri zostavovaní aplikácie ako aj samotné nasadenie s využitím *Azure IoT Hubu*.

5.1 Docker

Docker predstavuje open source platformu, ktorá uľahčuje vytváranie, nasadzovanie a spúšťanie aplikácií. Princíp spočíva v zabalení aplikácie spolu so všetkými závislosťami do izolovaného prostredia s názvom kontajner. To umožňuje vývojárom oddeliť aplikáciu od infraštruktúry, čo v konečnom dôsledku vedie k rýchlejšiemu dodaniu softvéru. [37]

5.1.1 Dockerfile

Pre spustenie aplikácie v Docker kontajneri, bolo najskôr potrebné vytvoriť vlastný obraz[14]. Vytvorenie obrazu prebieha na základe príkazov, ktoré sú uvedené v špeciálnom súbore s názvom *Dockerfile*. Súčasťou aplikácie sú Dockerfile súbory pre rôzne sady procesorov. Ukážka tohoto súboru pre *arm32* procesory je zobrazená vo Výpise 22.

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /source
RUN curl -sL https://deb.nodesource.com/setup_12.x | bash -
RUN apt-get install -y nodejs

COPY *.sln .
COPY Abb.CZOPC.WebService/*.csproj ./Abb.CZOPC.WebService/
COPY Abb.CZOPC.WebService.Common/*.csproj ./Abb.CZOPC.WebService.Common/
COPY Abb.CZOPC.WebService.DataAccess/*.csproj ./Abb.CZOPC.WebService.DataAccess
    /
COPY Abb.CZOPC.WebService.Entities/*.csproj ./Abb.CZOPC.WebService.Entities/
COPY Abb.CZOPC.WebService.Domain/*.csproj ./Abb.CZOPC.WebService.Domain/
COPY Abb.CZOPC.WebService.Database/*.csproj ./Abb.CZOPC.WebService.Database/
COPY Abb.CZOPC.WebService.Tests/*.csproj ./Abb.CZOPC.WebService.Tests/
RUN dotnet restore -r linux-arm

COPY . .
WORKDIR /source/Abb.CZOPC.WebService
RUN dotnet publish -c release -o /app -r linux-arm
```

[14]Docker obraz predstavuje základnú spustiteľnú jednotku, ktorú využívajú kontajnery pre svoj beh.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim-arm32v7
WORKDIR /app
COPY --from=build /app ./
EXPOSE 80 443
ENTRYPOINT ["./Abb.CZOPC.WebService"]
```

Výpis 22: Dockerfile súbor pre arm32 procesory

Základom je oficiálny obraz pre .NET Core aplikácie s názvom *.NET Core SDK*[15]. Tento obraz však neobsahuje Node.js prostredie pre build klientskej časti aplikácie a preto bolo potrebné vykonať jeho inštaláciu. Po inštalácií nevyhnutných súčastí dochádza k obnoveniu všetkých balíčkom a buildu aplikácie.

Skompilované súbory sú následne prekopírované do obrazu s názvom *ASP.NET Core Runtime*[16], ktorý slúži ako behové prostredie pre aplikáciu.

5.1.2 Docker Swarm

Jedná sa o nástroj, pomocou ktorého je možné vytvoriť viacero kontajnerov jediným príkazom. Konfigurácia kontajnerov prebieha v súbore s názvom *docker-compose.yml*. Podobne ako v prípade Dockerfile súborov bolo vytvorených viacero konfigurácií pre rôzne typy procesorov. Výpis 23 zobrazuje ukážku docker-compose.yml súboru pre arm32 procesory.

```
version: '3.0'
services:
  ws:
    image: robotwebservice.azurecr.io/robotwebservice:206-arm32v7
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
    networks:
      - wsnet
    depends_on:
      - postgres
  postgres:
    image: postgres
```

[15]Informácie o obraze: `https://hub.docker.com/_/microsoft-dotnet-core-sdk`

[16]Informácie o obraze: `https://hub.docker.com/_/microsoft-dotnet-core-aspnet`

```
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    environment:
      POSTGRES_PASSWORD: CzOPcWebSserviCe
      POSTGRES_USER: webservice
      POSTGRES_DB: CZOPC-WebService
    ports:
      - "5432:5432"
    networks:
      - wsnet
    volumes:
      - postgres-data:/var/lib/postgresql/data
networks:
  wsnet:
volumes:
  postgres-data:
```

Výpis 23: Docker-compose.yml súbor pre arm32 procesory

Súbor obsahuje konfiguráciu pre spustenie *Robot Web Service* aplikácie spoločne s databázou *PostgreSQL*[17]. Jednotlivé kontajnery obsahujú konfiguráciu portu, počtu replikácií, akcie po zlyhaní a v prípade databázy aj jej úvodné nastavenia. Kontajnery spolu komunikujú prostredníctvom siete s názvom *wsnet*.

Spustenie prebieha prostredníctvom príkazu *docker-compose up*. Tento príkaz sa prevažne používa v lokálnom vývojovom prostredí. Vypnutie umožňuje príkaz *docker-compose down*.

Alternatívu pre produkčné prostredie predstavuje príkaz *docker stack deploy -c docker-compose.yml "názov"*. V tomto prípade je nevyhnutné, aby hosť bežal v *swarm* móde. Tento mód sa zapína prostredníctvom príkazu *docker swarm init*. Po jeho zapnutí je možné prepájať viacero hostov, definovať kde a v akých počtoch sa majú nasadzovať jednotlivé kontajnery, poprípade akcie pri zlyhaní. Vypnutie prebieha prostredníctvom príkazu *docker stack rm "názov"*.

5.2 Azure IoT Hub

Azure IoT Hub predstavuje cloud službu, ktorá slúži ako centrálny bod pre správu aplikácií. Umožňuje vzdialené nasadzovanie Docker kontajnerov, ich aktualizáciu alebo monitorovanie. [38]

[17] Informácie o obraze: `https://hub.docker.com/_/postgres`

5.2.1 Pridanie zariadení

Aby bolo možné pripojiť a spravovať zariadenia prostredníctvom IoT Hubu, je potrebné vykonať nasledujúce kroky:

- Založenie služby IoT Hub[18].
- Pridanie zariadení prostredníctvom možnosti IoT Edge.
- Inštalácia podporných balíčkov na zariadenia pre komunikáciu s IoT Hubom[19].
- Nastavenie reťazcov pripojenia pre jednotlivé zariadenia.

Ďalej môžeme zariadenia logicky rozdeľovať pomocou *tagov*, čo nám dáva plnú kontrolu pri nasadzovaní kontajnerov. Môžeme definovať zariadenia pre testovanie, konkrétnych zákazníkov, poprípade operačné systémy a architektúry procesorov.

5.2.2 Konfigurácia aplikácie

Azure IoT Hub pracuje s pojmom modul. Modul predstavuje samostatnú aplikáciu, poprípade jej časť bežiacu v Docker kontajneri. Konfigurácia modulu prebieha v súbore s názvom *module.json*. Ukážku tohoto súboru môžete vidieť vo Výpise 24.

```
{
    "$schema-version": "0.0.1",
    "description": "",
    "image": {
        "repository": "${ACR_ADDRESS}/${ACR_IMAGE}",
        "tag": {
            "version": "1.0.0",
            "platforms": {
                "amd64": "./Dockerfile",
                "arm32v7": "./Dockerfile.arm32v7"
            }
        },
        "buildOptions": []
    },
    "language": "csharp"
}
```

Výpis 24: Ukážka súboru module.json

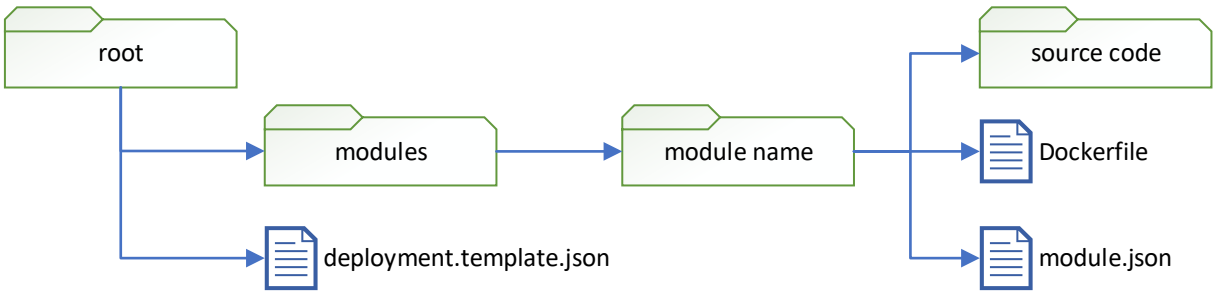
[18]Založenie: `https://docs.microsoft.com/en-us/azure/iot-edge/how-to-register-device`

[19]Inštalácia: `https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux`

Súbor obsahuje nastavenia, ktoré zabezpečia vytvorenie Docker obrazov pre rôzne typy procesorov a ich uloženie do definovaného repozitára.

Ďalší dôležitý súbor predstavuje *deployment.template.json*. Ten obsahuje konfiguráciu pre nasadenie modulov. Každý modul obsahuje definíciu obrazu, z ktorého má byť vytvorený, port na ktorom bude dostupný a ďalšie dodatočné nastavenia ako úložisko alebo akcia pri zlyhaní.

Zdrojové kódy ako aj súbory pre vytvorenie modulov a nasadenie aplikácie musia byť správne štruktúrované. Ukážka tejto štruktúry je zobrazená na Obrázku 53.



Obr. 53: Azure IoT Hub štruktúra projektu

5.3 Azure DevOps

Jedná sa o platformu, ktorá umožňuje verziovať zdrojové kódy, organizovať prácu na softvéri, vykonávať buildy a ďalšie činnosti spojené s konfiguračným managementom. [39]

5.3.1 Verziovanie kódu

Projekt využíva verziovací systém *Git*. Zdrojové kódy sú udržiavané v dvoch hlavných vetvách:

- **Master:** Stabilná verzia aplikácie pre produkčné prostredie.
- **Development:** Zdrojové kódy obsahujúce novú funkcionalitu a opravy.

Vývoj aplikácie prebiehal v malých prírastkových verziách. Jednotlivé funkcionality boli implementované v samostatných vetvách a po ich dokončení došlo k premietnutiu týchto zmien do Development vetvy prostredníctvom *pull requestu*.

5.3.2 Zostavenie

Build aplikácie prebieha prostredníctvom *Azure DevOps Pipelines*. Tie umožňujú automatizovať činnosti spojené s testovaním, zostavením a nasadením aplikácie. Nachádza sa tu podpora pre veľké množstvo technológií ako je Docker, Azure a podobne. [40]

Obe hlavé vývojové vetvy obsahujú vlastný build, ktorý je automaticky spúšťaný po dokončení pull requestu. Buildy pozostávajú z nasledujúcich krokov:

- **NuGet restore:** Inštalácia balíčkov pre projekt.

- **Test:** Spustenie unit testov a vytvorenie reportu pokrytia. Build pokračuje po úspešnom vykonaní všetkých testov.
- **File transform:** Navýšenie verzie Docker obrazu.
- **Run Code Analysis:** Spustenie SonarQube analýzy.
- **Build:** Samotný build zdrojových kódov.
- **Publish Quality Gate Result:** Odoslanie výsledkov analýzy do SonarQube nástroja.
- **Azure IoT Edge - Build module images:** Vytvorenie Docker obrazu na základe konfigurácií uvedených v súboroch *module.js* a *deployment.template.json*.
- **Azure IoT Edge - Push module images:** Nahratie vytvoreného obrazu do súkromného repozitára *Azure Container Registry*.
- **Copy Files:** Prekopírovanie súborov *module.js* a *deployment.template.json* do cieľového adresára.
- **Publish Artifact:** Vypublikovanie skopírovaných súborov pre potreby nasadenia.

5.3.3 Nasadenie

Po úspešnom builde aplikácie dochádza k jej nasadeniu. Táto akcia je taktiež spúšťaná automaticky a obsahuje nasledujúce kroky:

- **Azure IoT Edge - Generate deployment manifest:** Na základe súboru *deployment.template.json* dochádza k vytvoreniu finálneho konfiguračného súboru.
- **Azure IoT Edge - Deploy to IoT Edge devices:** Samotné nasadenie aplikácie na cieľové zariadenia. Umožňuje vykonať nasadenie na konkrétne zariadenie, poprípade množinu zariadení s využitím tagov.

Po úspešnom nasadení sú tie zmeny viditeľné v prostredí Azure IoT Hub. Nachádzajú sa tu informácie o jednotlivých moduloch, ako je stav alebo spôsob nasadenia. Ukážka je zobrazená na Obrázku 54.

NAME	TYPE	SPECIFIED IN DEPLOYMENT	REPORTED BY DEVICE	RUNTIME STATUS	EXIT CODE
$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes	running	0
$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes	running	0
postgres	IoT Edge Custom Mod...	✓ Yes	✓ Yes	running	0
robotwebservice	IoT Edge Custom Mod...	✓ Yes	✓ Yes	running	0

Obr. 54: Azure IoT Edge informácie o nasadených moduloch

Záver

Cieľom diplomovej práce bolo navrhnúť a implementovať softvérové riešenie pre zber dát z priemyselných robotov nezávisle na type monitorovanej platformy. Súčasťou práce bola taktiež analýza aktuálneho stavu, úprava existujúcich aplikácií a otestovanie výsledného riešenia pre použitie v produkčnom prostredí.

Ako prvá bola vykonaná analýza rozhrania pre komunikáciu s robotmi od spoločnosti ABB. Boli identifikované kľúčové vlastnosti, ako aj zásadné rozdiely medzi verziami 1.0 a 2.0. Ďalej boli popísané existujúce aplikácie a ich nedostatky (zabezpečenie, rozdielne verzie API rozhraní, prerušenie získavania dát, ukladanie dát).

Po dôkladnej analýze nasledoval návrh systému, ktorý by vyriešil vyššie spomenuté problémy. Návrh bol vykonaný z viacerých pohľadov (implementačný, logický, procesný, nasadenia, prípadov užitia), čo umožnilo vykonať rozbor od najvyššej úrovne abstrakcie, kedy boli popísané jednotlivé funkcionality systému, cez rozdelenie softvéru do jednotlivých vrstiev až po podrobný popis tried, ich vzájomných vzťahov a zodpovedností. V procese návrhu ďalej došlo k výberu vhodného zariadenia pre server (Raspberry Pi 3B+) a popisu technológií pre jednotivé časti aplikácie.

Na základe návrhu som pristúpil k samotnej implementácií webovej aplikácie, ktorá sa skladá zo serverovej a klientskej časti. Serverová časť je napísaná v jazyku C# a počas vývoja boli použité moderné techniky programovania, aby bolo možné aplikáciu jednoducho udržovať, testovať a rozširovať. Bol využitý princíp AOP pre logovanie, správa závislostí prostredníctvom IoC kontajnera, programovanie voči rozhraniu či SignalR protokol pre komunikáciu s klientmi v reálnom čase. Autentifikácia prebieha prostredníctvom JWT tokenu a dáta sú sprístupnené cez REST API rozhranie. Klientská časť aplikácie je napísaná v JavaScriptovom frameworku React s využitím jazyka TypeScript, ktorý ho rozširuje o statickú typovú kontrolu. Užívateľské rozhranie je v súlade s oficiálnym dizajnom spoločnosti ABB a optimalizované pre rôzne veľkosti obrazoviek. Užívateľ má k dispozícií prehľad priradených robotov, môže si zobraziť a filtrovať získané dáta, alarmy alebo priebeh komunikácie. Ďalej došlo k úprave iOS a tvOS aplikácií, aby dokázali komunikovať s vytvoreným serverom, a taktiež k presunu existujúcej HTML stránky z robota na stranu servera.

Súčasťou práce boli aj rozličné spôsoby testovania. Aplikácia obsahuje unit testy pre vybrané triedy z doménovej vrstvy, ktoré pomáhajú overiť funkčnosť jednotlivých metód pri zmenách implementácie. Ďalej bola vykonaná statická analýza kódu prostredníctvom nástroja SonarQube. Jej výstupom je zoznam bugov, zraniteľností, úroveň pokrytia kódu unit testami alebo množstvo duplikácií. Týmto spôsobom je možné odhaliť rôzne nedokonalosti v zdrojovom kóde a dopomôcť k jeho vyššej kvalite a bezpečnosti. Na záver bol systém ešte podrobený výkonnostnému testovaniu. Konkrétne sa jednalo o load, stress a SignalR testovanie. Výkonnostné testovanie mi pomohlo odhaliť úzke miesta výslednej aplikácie, ktoré boli spôsobené absenciou databázo-

vých indexov a neoptimálnym vykonávaním niektorých dotazov. Po ich odstránení server spĺňa požadovanú úroveň výkonu s dostatočnou rezervou.

Diplomová práca sa taktiež zaoberala nasadením a správou životného cyklu aplikácie. Pre tieto potreby boli využité technológie Docker, Azure DevOps, Azure Container Repository a Azure IoT Edge. Docker umožňuje beh aplikácie spolu s PostgreSQL databázou v prostredí kontajnerov. Výhoda tohoto riešenia spočíva v rýchlosti nasadenia a jednoduchom prenesení zo servera na server v prípade potreby. Výsledné riešenie počíta s dvoma prípadmi použitia. V prvom prípade nemá server prístup k internetu a kontajnery sú spúšťané manuálne v Docker Swarm móde. Pre tento účel bol vytvorený súbor s názvom *docker.compose.yml*, ktorý obsahuje všetku potrebnú konfiguráciu pre nasadenie kontajnerov. V prípade pripojenia na internet server komunikuje so službou Azure IoT Edge, ktorá umožňuje vzdialené nasadzovanie, monitorovanie a správu kontajnerov. Celý proces, od správy zdrojového kódu, cez testovanie a zostavenie aplikácie až po jej nasadenie je plne automatizovaný prostredníctvom platformy Azure DevOps. Tá sa taktiež stará o vytvorenie Docker obrazov a ich uloženie do súkromného repozitára Azure Container Repository. Každý obraz obsahuje číslo buildu aby bolo možné dohľadať vykonané zmeny.

Výstupom práce je plne funkčný nástroj pre monitorovanie robotov. Toto riešenie nielenže poskytuje užívateľom informácie o pripojených robotoch ale taktiež odstránilo problémy, ktorými trpia súčasné aplikácie a môže v budúcnosti dopomôcť k optimálnejšiemu využívaniu robotov na základe zozbieraných dát. V prípade iOS a tvOS aplikácií došlo k zmene spôsobu komunikácie. Aplikácie teraz komunikujú so serverom namiesto priameho pripojenia na robotov. Tým sa odstránili problémy s používaním aplikácií bez vedomia vlastníka, neschopnosť získavať dáta z robotov s novým typom RobotWaru a v neposlednom rade pády robotov do núdzového režimu. Zmenami si taktiež prešla existujúca HTML stránka, ktorá je teraz súčasťou servera a tým sa odstránila potreba hosťovania tejto stránky priamo na robotovi. V konečnom dôsledku viedli všetky tieto zmeny k nižšiemu počtu dotazov na robota. Počas testovania nedošlo k žiadnemu pádu robota do núdzového režimu a ani pozastaveniu získavania dát z dôvodu častého dotazovania. Bolo zaznamenaných zopár výpadkov spojenia, ktoré však boli spôsobené problémami so sieťou a vždy došlo k opätovnému nadviazaniu komunikácie. V diplomovej práci vidím veľký potenciál v budúcom rozširovaní a využití v praxi.

Literatúra

[1] developercenter.robotstudio.com, *An introduction to Robot Web Services* [online] [cit. 2019-10-06], dostupné z: http://developercenter.robotstudio.com/webservice/api_reference

[2] developercenter.robotstudio.com, *An introduction to Robot Web Services* [online] [cit. 2019-10-06], dostupné z: http://developercenter.robotstudio.com/templates/swagger.html

[3] CHESHIRE, Stuart a Daniel H. STEINBERG. *Zero configuration networking: the definitive guide*. Sebastopol, CA: O'Reilly, 2006. ISBN 0596101007

[4] cs.ubc.ca, *Architectural Blueprints—The "4+1" View Model of Software Architecture* [online] [cit. 2019-12-8], dostupné z: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

[5] library.e.abb.com, *ABB Ability^{TM} Connected Services* [online] [cit. 2019-12-10], dostupné z: https://library.e.abb.com/public/fe588818c4c14fae8b42863aad0047ec/Oct%2031st%20ConnectedServicesDatasheet.pdf?x-sign=7GZaebZbBLi3mqJrvhkEBiKt2IGLnsUZQM3oB79VBcwFoLsPnRkoW9CocX4qukdG

[6] connect.kuka.com, *KUKA Connect Administrative Guide* [online] [cit. 2019-12-14], dostupné z: https://kuka-atx-public-assets.s3-us-west-2.amazonaws.com/imagery/documentation/downloads/KUKA_Connect_Administrative_Guide_EN.pdf

[7] connect.kuka.com, *KUKA Connect Features and Functionalities Guide* [online] [cit. 2019-12-14], dostupné z: https://kuka-atx-public-assets.s3-us-west-2.amazonaws.com/imagery/documentation/downloads/KUKA_Connect_Features_Functionalities_Guide_EN.pdf

[8] docs.microsoft.com, *What is Nuget?* [online] [cit. 2020-03-22], dostupné z: https://docs.microsoft.com/en-us/nuget/what-is-nuget

[9] autofaccn.readthedocs.io, *Getting Started* [online] [cit. 2020-03-22], dostupné z: https://autofaccn.readthedocs.io/en/latest/getting-started/index.html

[10] docs.automapper.org, *Getting Started Guide* [online] [cit. 2020-03-22], dostupné z: http://docs.automapper.org/en/stable/Getting-started.html

[11] dbup.readthedocs.io, *Home - DbUp* [online] [cit. 2020-03-22], dostupné z: https://dbup.readthedocs.io/en/latest/#getting-started

[12] dapper-tutorial.net, *Dapper* [online] [cit. 2020-03-22], dostupné z: https://dapper-tutorial.net/dapper

[13] docs.microsoft.com, *Introduction to ASP.NET Core SignalR* [online] [cit. 2020-03-23], dostupné z: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-3.1

[14] github.com, *Tmds.MDns* [online] [cit. 2020-03-24], dostupné z: https://github.com/tmds/Tmds.MDns

[15] html-agility-pack.net, *Html Agility Pack* [online] [cit. 2020-03-25], dostupné z: https://html-agility-pack.net

[16] docs.npmjs.com, *About npm* [online] [cit. 2020-03-26], dostupné z: https://docs.npmjs.com/about-npm

[17] redux.js.org, *Three Principles* [online] [cit. 2020-03-26], dostupné z: https://redux.js.org/introduction/three-principles

[18] ant.design, *Ant Design of React* [online] [cit. 2020-03-26], dostupné z: https://ant.design/docs/react/introduce

[19] github.com, *Axios* [online] [cit. 2020-03-26], dostupné z: https://github.com/axios/axios

[20] sass-lang.com, *Documentation* [online] [cit. 2020-03-26], dostupné z: https://sass-lang.com/documentation

[21] docs.microsoft.com, *ASP.NET Core SignalR JavaScript client* [online] [cit. 2020-03-26], dostupné z: https://docs.microsoft.com/en-us/aspnet/core/signalr/javascript-client?view=aspnetcore-3.1

[22] guides.cocoapods.org, *Getting Started* [online] [cit. 2020-03-28], dostupné z: https://guides.cocoapods.org/using/getting-started.html

[23] github.com, *Alamofire* [online] [cit. 2020-03-28], dostupné z: https://github.com/Alamofire/Alamofire

[24] github.com, *Keychain-swift* [online] [cit. 2020-03-28], dostupné z: https://github.com/evgenyneu/keychain-swift

[25] github.com, *SwiftyJSON* [online] [cit. 2020-03-28], dostupné z: https://github.com/SwiftyJSON/SwiftyJSON#why-is-the-typical-json-handling-in-swift-not-good

[26] github.com, *SwiftSignalRClient* [online] [cit. 2020-03-28], dostupné z: https://github.com/moozzyk/SignalR-Client-Swift

[27] docs.realm.io, *What is Realm Platform?* [online] [cit. 2020-03-28], dostupné z: https://docs.realm.io/sync/what-is-realm-platform

[28] MYERS, Glenford J., Corey SANDLER a Tom BADGETT. *The art of software testing*. 3rd ed. Hoboken, N.J.: John Wiley, c2012. ISBN 9781118031964.

[29] github.com, *Coverlet* [online] [cit. 2020-03-29], dostupné z: `https://github.com/tonerdo/coverlet`

[30] github.com, *Moq* [online] [cit. 2020-03-29], dostupné z: `https://github.com/moq/moq4`

[31] github.com, *NUnit 3 Framework* [online] [cit. 2020-03-30], dostupné z: `https://github.com/nunit/nunit`

[32] MOLYNEAUX, Ian. *The art of application performance testing*. Second edition. Beijing: O'Reilly, 2014. ISBN 9781491900543.

[33] jmeter.apache.org, *Apache JMeter™* [online] [cit. 2020-03-30], dostupné z: `https://jmeter.apache.org/index.html`

[34] github.com, *Crankier* [online] [cit. 2020-03-31], dostupné z: `https://github.com/dotnet/aspnetcore/tree/master/src/SignalR/perf/benchmarkapps/Crankier`

[35] searchwindevelopment.techtarget.com, *Static analysis (static code analysis)* [online] [cit. 2020-04-09], dostupné z: `https://searchwindevelopment.techtarget.com/definition/static-analysis`

[36] sonarqube.org, *Code Quality and Security* [online] [cit. 2020-04-09], dostupné z: `https://www.sonarqube.org`

[37] docs.docker.com, *Docker overview* [online] [cit. 2020-04-13], dostupné z: `https://docs.docker.com/get-started/overview`

[38] docs.microsoft.com, *What is Azure IoT Hub?* [online] [cit. 2020-04-21], dostupné z: `https://docs.microsoft.com/en-us/azure/iot-hub/about-iot-hub`

[39] docs.microsoft.com, *What is Azure DevOps?* [online] [cit. 2020-04-21], dostupné z: `https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops`

[40] docs.microsoft.com, *What is Azure Pipelines?* [online] [cit. 2020-04-24], dostupné z: `https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops`

A Postup spustenia

Pre spustenie výsledného riešenia je potrebné mať na svojom operačnom systéme nainštalovanú virtualizačnú platformu Docker[20] spolu s nástrojom Docker Compose[21]. Následne je možné pokračovať týmito krokmi:

- Spustenie príkazového riadku v priečinku s cestou: *Implementácia/Server/Docker/Dev*
- Zadanie príkazu *docker-compose up*
  - V prípade obsadenosti portov 80 alebo 5432, je potrebné prestaviť tieto porty v súbore *docker-compose.yml*
- Otvorenie aplikácie na adrese: *http://localhost:"port"*
- Nastavenie hesla pre preddefinovaného užívateľa *admin* v časti *Reset password*
- Pre url adresy robotov ma kontaktuje na email: *jakub.kacik.st@vsb.cz*

V priečinku s názvom *Databáza* sa nachádza záloha databázy so zozbieranými údajmi z robotov za obdobie od 01.04.2020 do 14.04.2020. Obnovenie je možné vykonať prostredníctvom nástroja *PgAdmin* a zahŕňa nasledujúce kroky:

- Zmazanie pôvodnej databázy
- Vytvorenie novej databázy pomocou príkazov uvedených v súbore *Init.sql*
- Obnovenie databázy zo zálohy[22]
- Záloha obsahuje prednastavené užívateľské účty *admin* a *user* - heslo je zhodné s názvom užívateľa

Aplikáciu je taktiež možné spustiť prostredníctvom vývojového nástroja *Visual Studio*. Zdrojové kódy sa nachádzajú v priečinku s cestou: *Implementácia/Server/modules/RobotWebService*. Pred spustením je potrebné vytvoriť PostgreSQL databázu príkazmi zo súboru *Init.sql*. Aplikácia komunikuje s databázou prostredníctvom portu 5432. Toto nastavenie je možné zmeniť v súbore *appsettings.Development.json*, ktorý sa nachádza v projekte s názvom *Abb.CZOPC.WebService*.

Na adrese *http://rws.northeurope.cloudapp.azure.com* aktuálne beží testovacia prevádzka aplikácie. Pre získanie prihlasovacích údajov ma kontaktuje na email *jakub.kacik.st@vsb.cz*.

[20] Docker inštalácia pre Ubuntu: `https://docs.docker.com/engine/install/ubuntu`

[21] Docker Compose inštalácia pre Linux: `https://docs.docker.com/compose/install`

[22] Obnovenie databázy: `https://www.pgadmin.org/docs/pgadmin4/development/restore_dialog.html`